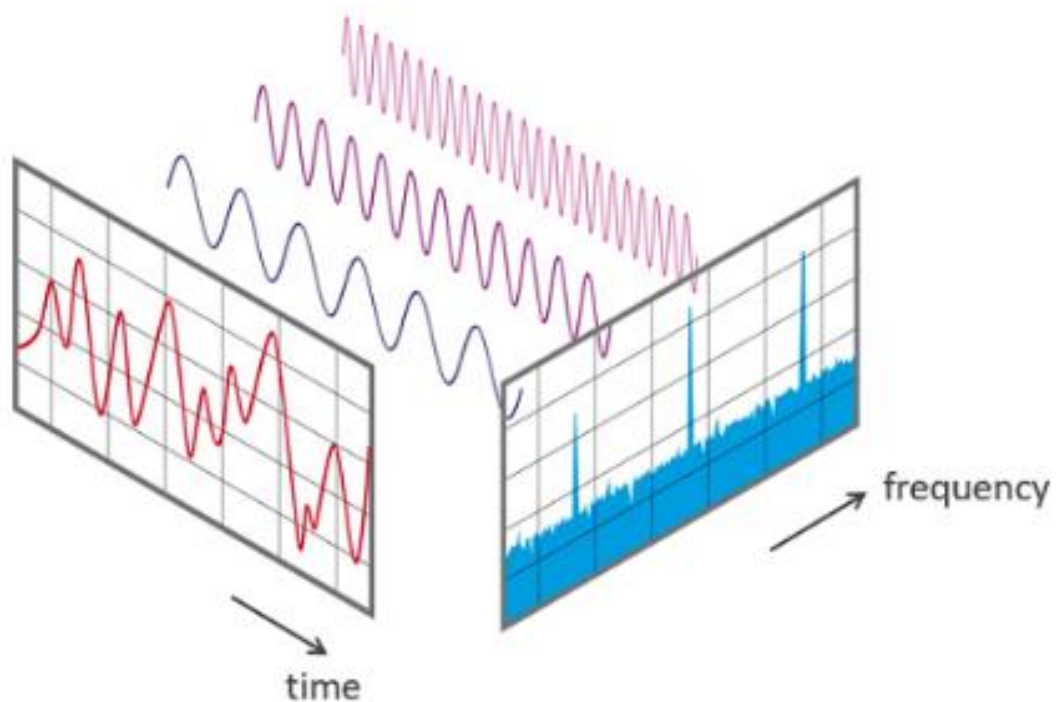


---

# CAB401 - PARALLELIZATION PROJECT

---

Parallel Signal Processing



Samuel Rieger – n10773967  
27/10/23

## Contents

Program Breakdown .....	2
Background Research.....	3
Tools & Environment .....	3
Libraries & Namespaces.....	3
Specifications .....	3
Overall Initial Analysis.....	4
<i>timefreq.cs</i> Optimisation .....	5
Parallelisation.....	5
<i>onsetDetection()</i> Optimisation.....	6
Parallelisation.....	6
Challenges .....	7
<i>fft()</i> Optimisation .....	7
Restructuring.....	7
Parallelisation.....	8
Asynchronous Optimisation.....	9
Verifying Parallel Output.....	10
Results & Analysis .....	10
Compile & Run Instructions .....	12
Conclusion.....	12
References .....	12
Appendices.....	13
Appendix 1 .....	13
Appendix 2 .....	15

## Program Breakdown

For this project I have selected to parallelize and optimise the Digital Music Analysis C# application. The application is designed to give a music student (in this case a violin student) without a teacher, feedback on a recording of a song that they have played. It can tell whether each note played was sharp or flat (out of tune) and if the notes were played for the correct length of time. The application takes two inputs, an XML file for a piece of sheet music and a WAV file recording of the piece being played. The application uses Xamarin, a popular C# package to display a GUI for the user complete with three tabs. The first is a visualisation image of the time-frequency, the second is a histogram of the note frequencies for each octave and the third is a stave which gives the user feedback on where they went wrong while playing the song.

This application applies a Fourier Transform to the audio WAV file which unpacks the sampled signal back into its original individual frequencies. This is useful as each note in music has a certain specific frequency that the instrument or violin strings in this case need to resonate at to play the desired note. Breaking down a signal back into the frequencies that make it up and comparing them with the original music's note frequencies will determine whether the right note is being played or not.

The program consists of four classes as can be seen in the basic class diagram in figure 1 (note only important methods are shown for *MainWindow*). The classes *musicNote*, *noteGraph* and *wavefile* serve to store and visualise music data whereas *MainWindow* controls the GUI of the application. The class *timefreq* is dedicated to frequency analysis and is called from the method *onsetDetection()* in *MainWindow*. The *fft()* method in both *MainWindow* and *timefreq* does the Fourier Transform and is called from *onsetDetection()* and *stft()* respectively (visualised in figure 1).

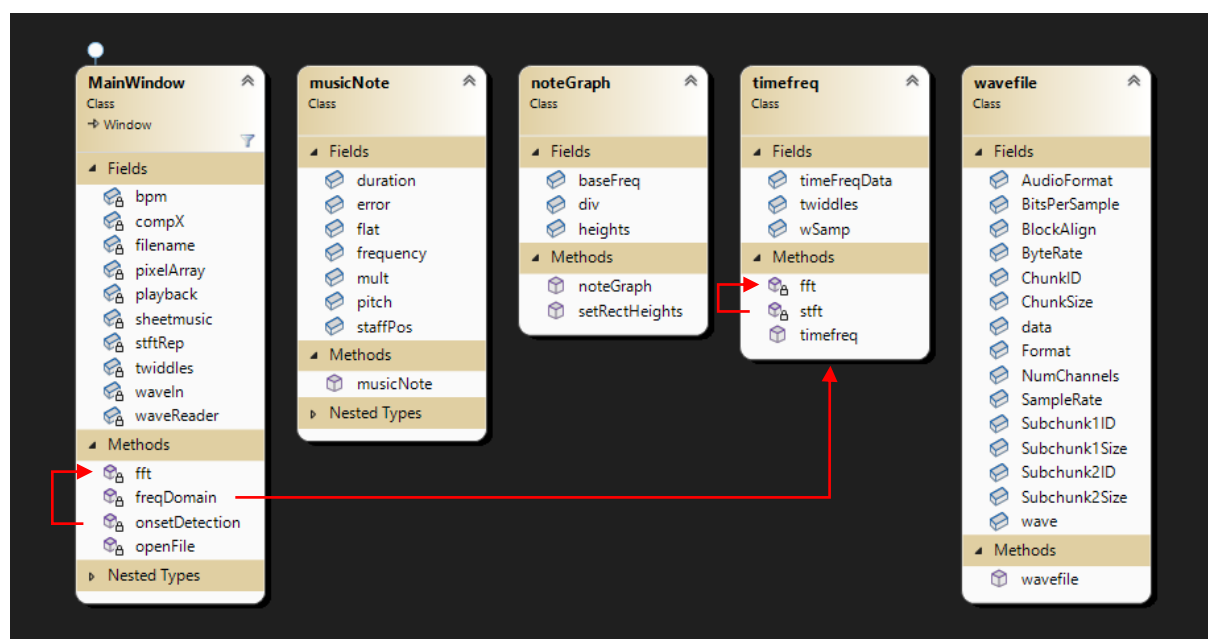


Figure 1

## Background Research

This application uses the terms 'fft' and 'stft' as a name for some of its algorithmic methods which also happen to use a large portion of computing resources. These are acronyms for famous algorithms used in Fourier Transform signal processing namely the Fast Fourier Transform (FFT) and Short Time Fourier Transform (STFT). While an explanation of how these algorithms work is beyond the scope of this report, researching and understanding how these algorithms work was essential to informing the approach taken for optimization and parallelization. Some high-level explanations of how FFT, STFT and other related algorithms work will be demonstrated throughout the report to support my methodology.

## Tools & Environment

The computer I will be using to run the program and perform parallelization testing on is my personal home desktop Windows computer. It is important that this computer runs on a Windows operating system as the program is written in C# and a Windows system natively runs Visual Studio unlike Mac OS which runs a ported cutdown version. Visual Studio is essential for debugging and performance profiling the sequential application.

Since the application is written in C# the parallel version of the application uses a few extra .NET libraries and namespaces for various debugging, profiling and of course parallelization purposes.

### Libraries & Namespaces

**System.Diagnostics:** Used for printing to the console for debugging purposes and timing various parts of the application for performance profiling purposes.

**System.Threading & Task Parallel Library (TPL):** Used for creating and running threads in parallel. Thread objects have useful methods such as *Join()* which ensures synchronization can be achieved. TPL also gives access to other implicit thread creation methods such as *Parallel.For()*.

### Specifications

**OS:** Windows 10 Pro

**System Type:** 64-bit operating system, x64-based processor

**Processor:** Intel Core i7-10700k (@ 3.80GHz, 8 Cores, 16 threads)

**RAM:** 16 GB

**GPU:** Nvidia (Not applicable for this project)

**IDE & Code Editor:** Visual Studio 2022

## Overall Initial Analysis

Upon performance profiling the original sequential program we can get an overview of how the CPU is being used throughout the runtime. Figure 2 shows a breakdown of the program.

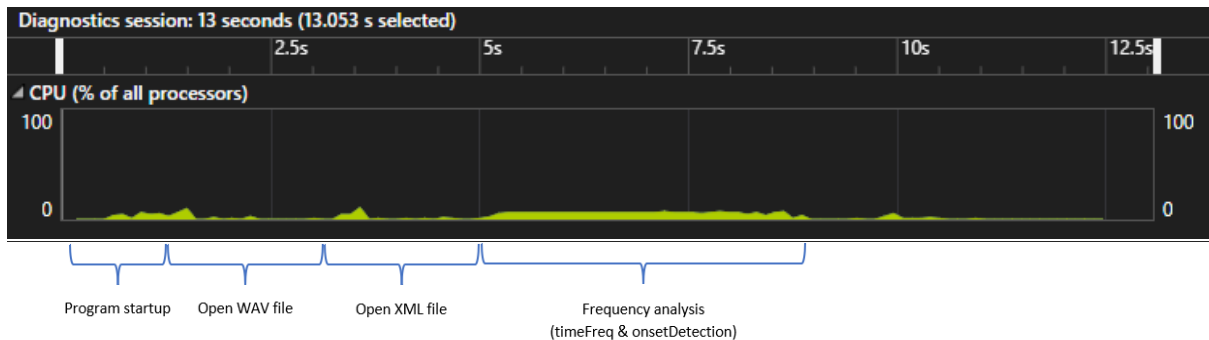


Figure 2

When highlighting the frequency analysis section of the runtime as shown in figure 3, we can see that there are two main methods taking up most of the CPU power. These being *timefreq.fft()* which is called from *timefreq.stft()* and *MainWindow.fft()* which is called from *MainWindow.onsetDetection()*.

Top Functions			
Function Name	Total CPU [unit, %]	Self CPU [unit, %]	
DigitalMusicAnalysis.timefreq.fft(System.Numerics.Complex[])	1533 (48.24%)	1365 (42.95%)	
DigitalMusicAnalysis.MainWindow.fft(System.Numerics.Complex[], int)	1064 (33.48%)	913 (28.73%)	
DigitalMusicAnalysis.MainWindow.onsetDetection()	1445 (45.47%)	183 (5.76%)	
[External Call] System.Numerics.Complex.op_Addition(System.Numerics.Complex, System.Numerics.Complex)	166 (5.22%)	166 (5.22%)	
[External Call] System.Numerics.Complex.op_Multiply(System.Numerics.Complex, System.Numerics.Complex)	139 (4.37%)	139 (4.37%)	

Figure 3

Therefore, *MainWindow.onsetDetection()* and the *timefreq* class is where most efforts of analysis will be focused for potential areas of parallelization and optimization. Upon further inspection the *fft()* method that is called in both classes is almost identical meaning optimizations that can be applied to one should be able to transfer to the other. This method could also be moved into its own class to cut down on duplicate code.

One other area that can be focused on for potential parallelization is the *openFile()* method found in *MainWindow*. This calls on an asynchronous method *ShowDialog()* which holds the program until a user has selected a file. Frequency analysis could be run in parallel after the first WAV file is selected to cut down on the perceived hang time before the main application window opens.

## timefreq.cs Optimisation

### Parallelisation

As observed in the call tree breakdown of the frequency analysis section shown in figure 4 *fft()* is called for *stft()* which uses a large percentage of CPU compute power. The algorithm which actually calculates and applies the Fourier Transform is the *fft()* method but why is it called from within *stft()*.

Function Name	Total CPU [unit, %]	Self CPU [unit, %]
▲ DigitalMusicAnalysis (PID: 15740)	3178 (100.00%)	35 (1.10%)
▲ DigitalMusicAnalysis.App.Main()	3142 (98.87%)	0 (0.00%)
▲ [External Call] presentationframework.dll!0x00007ffb8c55d78c	3142 (98.87%)	0 (0.00%)
▲ DigitalMusicAnalysis.MainWindow.ctor()	3142 (98.87%)	7 (0.22%)
▲ DigitalMusicAnalysis.MainWindow.freqDomain()	1647 (51.83%)	14 (0.44%)
▲ DigitalMusicAnalysis.timefreq.ctor(float32[], int)	1633 (51.38%)	14 (0.44%)
▲ DigitalMusicAnalysis.timefreq.stft(System.Numerics.Complex[], int)	1619 (50.94%)	79 (2.49%)
▶ DigitalMusicAnalysis.timefreq.fft(System.Numerics.Complex[])	1533 (48.24%)	82 (2.58%)

Figure 4

The *stft()* method partitions the original signal into separate frames with a defined sample length where each frame is passed onto the *fft()* method to calculate its Fourier Transform after which all frames are combined together in chronological order. This is because “STFT provides the time-localized frequency information for situations in which frequency components of a signal vary over time, whereas the standard Fourier transform provides the frequency information averaged over the entire signal time interval” (Nasser, 2008). Hence applying STFT makes sense as music is made up of multiple defined strong signals which change over time. If FFT was applied to the entire recording only frequencies that reoccurred the most would be prominent in the output.

However, this conveniently provides a great opening for parallelization. Since FFT is applied to each frame individually and is not dependent on any other FFT operations the Fourier Transform for each frame can be calculated in parallel. The method *fft()* only depends on the input sample frame (*x*) and *twiddles*, which is a complex array of multiplying factors used in the calculation. My approach for parallelization here was to split the *stft()* method up into two new methods. One called *distributeSTFT()* which partitions the frames into subsets and parses those partitions onto a cutdown *stft()* method running on its own thread. From here *stft()* parses each one of frames onto *fft()* to be processed. See figure 5 for a snippet of the sequential *stft()* and figure 6 for a snippet of the parallelized version. The full methods can be found in appendix 1.

```

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];
        }
    }
}

```

Figure 5

```

Thread[] stftThreads = new Thread[MainWindow.availableThreadNum];
numFramesPerThread = ((int)Math.Ceiling(2 * (double)N / (double)wSamp) + MainWindow.availableThreadNum) / (MainWindow.availableThreadNum);

for (int j = 0; j < stftThreads.Length; j++)
{
    stftThreads[j] = new Thread(stft);
    stftThreads[j].Start(j);
}

for (int j = 0; j < stftThreads.Length; j++)
{
    stftThreads[j].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;
    }
}

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

for (int k = startIndex; k < endIndex; k++)
{
    for (int j = 0; j < wSamp; j++)
    {
        temp[j] = xThread[k * (wSamp / 2) + j];
    }

    tempFFT = FourierTransform.fft(temp, twiddles);

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

        if (Y[j][k] > fftMax)
        {
            fftMax = Y[j][k];
        }
    }
}

```

Figure 6

As can be seen in figure 6 each thread executes its *Join()* method after it has been started so that synchronization of the parallel section is achieved.

These modifications can be found in lines 95 – 174 of *timefreq*.

## *onsetDetection()* Optimisation

### Parallelisation

When performance profiling *onsetDetection()*, it is evident that most of the work is done within the ‘for int mm’ loop of this sequential method (see figure 7).

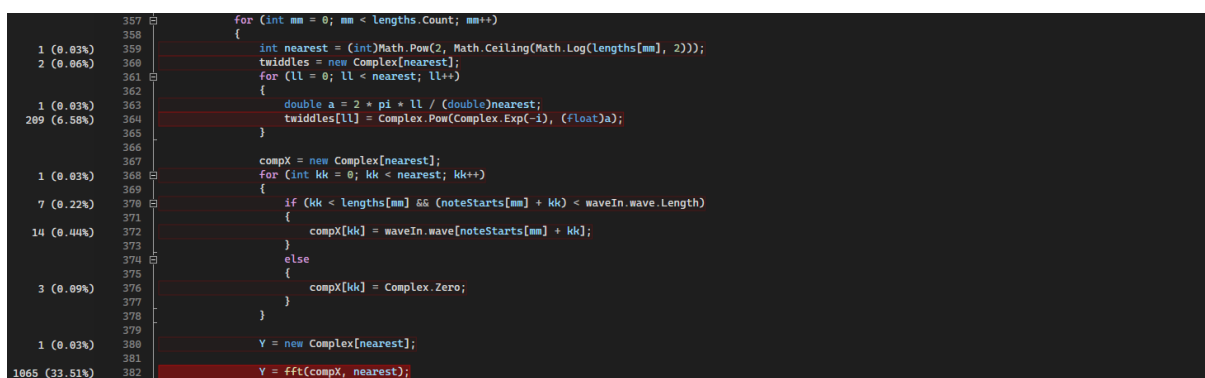


Figure 7

However, upon further analysis this loop can be broken up and restructured into multiple separate sections with two operations taking up most of the compute power – these being calculating the *twiddles* and *fft()* values. My approach here (which can be seen in figure 8) was to break it up into four separate loops and store calculated values that later sections were dependent on. The first

section calculates the *twiddles*, the second calculates the *compX* array and the third calculates the *fft()* which is dependent on both the *twiddles* and *compX* arrays, using them as inputs. The fourth loop does the remainder of the operations using the *fft()* outputs. Because only the *twiddles* and *fft()* loops used a large amount of CPU power these were the only loops that saw benefit in parallelization.

```
// Calculate Twiddles
Parallel.For(0, lengths.Count, parallelOptions, mm =>
{
    for (int ll = 0; ll < nearestArray[mm]; ll++)
    {
        double a = 2 * pi * ll / (double)nearestArray[mm];
        twiddlesArray[mm][ll] = Complex.Pow(Complex.Exp(-i), (float)a);
    }
});

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

// Calculate FFT
Parallel.For(0, lengths.Count, parallelOptions, mm =>
{
    VArray[mm] = FourierTransform.fft(compXArray[mm], twiddlesArray[mm]);
});
```

Figure 8

These modifications can be found in lines 389 – 434 of *MainWindow*.

## Challenges

While trying to parallelise the *twiddles* loop, I ran into quite a bit of trouble attempting to get a correct output and would often run into an ‘Index out of range’ exception. This was because in the sequential version the integer variable *ll* is defined when *onsetDetection()* is first called. Because of the loops dependency on *ll* the integer was being incremented multiple times by each instance of the loop which was communicated to each thread via their shared memory. This involved a simple fix of removing the definition from the start of the method and redefining a new *ll* each time the loop was run.

## fft() Optimisation

### Restructuring

Because *fft()* is used by both *MainWindow* and *timefreq* being virtually identical in both this can be restructured into its own class (*FourierTransform*) which is shared by both classes. The new *FourierTransform.fft()* takes two arguments *Complex[] x* and *Complex[] twiddles* where *x* is the sample frame and *twiddles* are the multiplying factors. While the original sequential program in *MainWindow* passes in *nearest* as an argument this is unnecessary as it is simply the frame size and can be derived from the sample frame array using *Complex[].Length*.

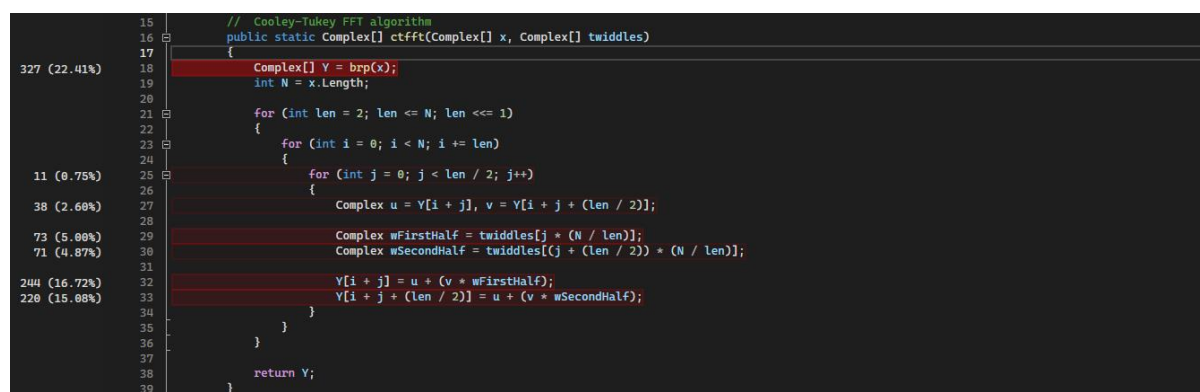
Restructuring the algorithm was the next obvious step after moving the *fft()* to its own function. This is where understanding and researching Fourier Transform algorithms was crucial. My approach here was to research an iterative version of the FFT algorithm as the recursive FFT was not able to be evenly broken up and run concurrently. My thinking here was that finding an iterative version of the FFT would expose some loop that I could exploit for parallelism. While FFT was considered a quick Fourier Transform algorithm when it was first discovered in 1965 it has since been super ceded by a



more efficient version – this is where Cooley – Tukey's FFT algorithm comes in. Cooley – Tukey's FFT is the most common Fourier Transform used today and reduces FFT's time complexity of  $O(n^2)$  down to  $O(n \log n)$  which is a huge improvement even before parallelization. Fourier Transform algorithms basically work by reordering the samples as can be seen with FFT where it subdivides samples down until the transform is applied to a sample of length 1. Cooley – Tukey's takes advantage of the fact that samples are always reordered in a particular way, specifically in the order of a bit reversal permutation. The algorithm works in two parts. First the sample array is subject to the bit reversal permutation. This essentially involves reversing the bits that represent the index of every array element, assigning them to the new position defined by the reversed bits. After this the bit reversed sample is iterated through calculating the Fourier Transform for both the first  $N/2$  samples and last  $N/2$  samples at the same time.

### Parallelisation

Unfortunately, the Fourier Transform part of Cooley – Tukey's FFT does not see a significant performance improvement from parallelization often achieving the opposite effect by clogging up the Task Parallel Libraries thread que in testing. However, as seen in figure 9 in line 18 the bit reversal permutation method has a significant amount of granularity that can be exploited for parallelization.



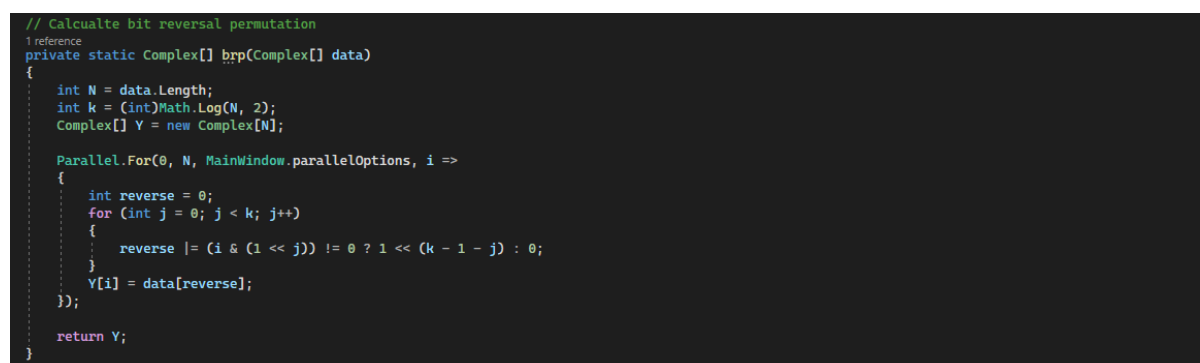
```

15 // Cooley-Tukey FFT algorithm
16 public static Complex[] ctfft(Complex[] x, Complex[] twiddles)
17 {
18     Complex[] Y = brp(x);
19     int N = x.Length;
20
21     for (int len = 2; len <= N; len <= 1)
22     {
23         for (int i = 0; i < N; i += len)
24         {
25             for (int j = 0; j < len / 2; j++)
26             {
27                 Complex u = Y[i + j], v = Y[i + j + (len / 2)];
28
29                 Complex wFirstHalf = twiddles[j * (N / len)];
30                 Complex wSecondHalf = twiddles[(j + (len / 2)) * (N / len)];
31
32                 Y[i + j] = u + (v * wFirstHalf);
33                 Y[i + j + (len / 2)] = u + (v * wSecondHalf);
34             }
35         }
36     }
37
38     return Y;
39 }

```

Figure 9

This is a simple parallelization as the loop is not dependent on any variables and simply iterates through each index of the sample frame assigning the sample to its reversed index. See figure 10 for a screenshot of the parallel *brp()* method.



```

// Calculate bit reversal permutation
1 reference
private static Complex[] brp(Complex[] data)
{
    int N = data.Length;
    int k = (int)Math.Log(N, 2);
    Complex[] Y = new Complex[N];

    Parallel.For(0, N, MainWindow.parallelOptions, i =>
    {
        int reverse = 0;
        for (int j = 0; j < k; j++)
        {
            reverse |= (i & (1 << j)) != 0 ? 1 << (k - 1 - j) : 0;
        }
        Y[i] = data[reverse];
    });

    return Y;
}

```

Figure 10

## Asynchronous Optimisation

The final major parallel exploitation can be found in the asynchronous *openFile()* selection method performed right at the beginning of the program in the *MainWindow* class. The methodology here was to run *openFile()* to select the WAV file first. Afterwards *freqDomain()* could be executed to calculate the Fourier Transform for the audio sample concurrently with the user selecting the XML file. Due to the *ShowDialog()* method used to open the file selection dialog being asynchronous awaiting the user selecting a file, *freqDomain()* will almost always complete its calculations before the user has made their selection. While the execution time of *freqDomain()* is unchanged the perceived execution time from the users perspective is just a little bit faster as the hang time after both files have been selected is reduced. For parallelizing this section, I once again used the Task Parallel Library this time utilizing its *Parallel.Invoke()* method. This method once again ensures synchronization which is imperative here as *onsetDetection()* cannot be executed without the XML data. See figure 11 for the sequential and parallel versions of the file selection section.

```
InitializeComponent();
filename = openFile("Select Audio (wav) file");
string xmlfile = openFile("Select Score (xml) file");
Thread check = new Thread(new ThreadStart(updateSlider));
loadWave(filename);
freqDomain();
sheetmusic = readXML(xmlfile);
onsetDetection();
loadImage();
loadHistogram();
playBack();
check.Start();
```

```
// Async Parallel
InitializeComponent();
filename = openFile("Select Audio (wav) file");
string xmlfile = "";
Parallel.Invoke(
    () =>
    {
        xmlfile = openFile("Select Score (xml) file");
    },
    () =>
    {
        loadWave(filename);
        freqDomain();
    }
);
Thread check = new Thread(new ThreadStart(updateSlider));
sheetmusic = readXML(xmlfile);
onsetDetection();
loadImage();
loadHistogram();
playBack();
check.Start();
```

Figure 11

## Verifying Parallel Output

To verify that the new parallel versions of methods produced the same results the Fourier Transform *float[]* array returned in both the sequential *stft()* and *onsetDetection()* methods was saved to a binary file and checked using the approach shown in figure 12. This proved to be an effective method of verifying the parallel results.

```
// Save timeFreqData as into a binary file.
using (Stream stream = File.Open("TimeFreqData.bin", FileMode.Create))
{
    BinaryFormatter bformatter = new BinaryFormatter();
    bformatter.Serialize(stream, timeFreqData);
}

// Check that the generated timeFreqData matches the benchmark one.
float[][] checkTimeFreq;

using (Stream stream = File.Open("TimeFreqData.bin", FileMode.Open))
{
    BinaryFormatter bformatter = new BinaryFormatter();
    checkTimeFreq = (float[][])bformatter.Deserialize(stream);
    for (int qq = 0; qq < checkTimeFreq.Length; qq++)
    {
        for (int ww = 0; ww < checkTimeFreq[qq].Length; ww++)
        {
            Debug.Assert(checkTimeFreq[qq][ww] == timeFreqData[qq][ww], "Incorrect timeFreqData produced.");
        }
    }
}
```

Figure 12

## Results & Analysis

As previously discussed, to measure run times of methods the *System.Diagnostics* namespace was used. This gives access to the *Stopwatch* object which can measure elapsed times in milliseconds. When mentioning the 'Overall' time that the application has run for this specifically refers to the time between the application starting and the *MainWindow* GUI opening for the user. It also excludes any *openFile()* methods as the timings of these can vary due to the users input. An example of how the timings for the 'Overall' application is calculated is given in appendix 2.

Since optimizing the *fft()* method it was refactored into a faster sequential version of the FFT algorithm. As such this is the new benchmark for the best sequential program.

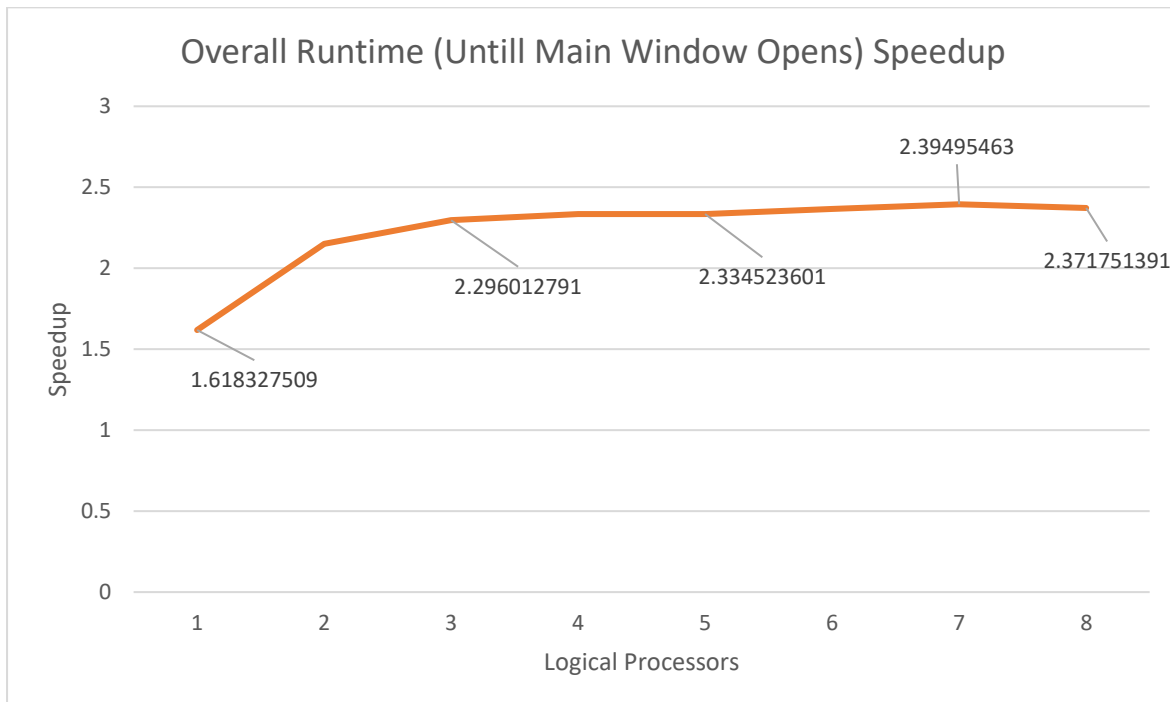
The following table shows the best sequential times for operations versus the best parallel times.

Comparison of Run Times (All times are an average of 10)			
Best Sequential Timings		Best Parallel Timings (14 threads)	
Method	Time (ms)	Method	Time (ms)
Overall	2580.475	Overall	1077.463
stft()	1108.883	<i>distributeSTFT()</i>	373.187
<i>onsetDetection()</i>	1077.227	<i>onsetDetection()</i>	309.609

As can be seen from this table the runtime is significantly improved across both the targeted methods and consequently the overall run time.

The following table shows a breakdown of run time when considering the number of logical processors. This is followed by a speedup chart for the overall runtime. It is important to note that in the case of this system there are two threads per core.

Parallel Method Run Times on Increasing Core Numbers (All times are an average of 10 in milliseconds)			
Cores (Threads)	Overall	<i>distributeSTFT()</i>	<i>onsetDetection()</i>
1 (2)	1594.532	557.460	613.742
2 (4)	1199.688	393.899	428.596
3 (6)	1123.894	378.783	378.630
4 (8)	1105.473	371.685	345.731
5 (10)	1105.354	366.876	327.700
6 (12)	1090.779	372.616	311.859
7 (14)	1077.463	373.187	309.609
8 (16)	1088.004	377.955	315.225



As can be seen from the speedup chart “perfect” linear speedup is not achieved however sublinear speedup is. Performance increase flattens out at around 3 cores and as 8 or more are used the run time actually starts to increase again.

I believe the reason behind the increased run time as 8 or more cores are used is due to the overhead of assigning tasks to threads. If the number of tasks per thread is relatively small, it does not make sense to increase the number of threads as this will produce more work than is necessary. With larger test audio samples this graph might look a bit more linear as more samples means more tasks to be distributed amongst the available number of threads. However, with only one test case this theory was not able to be verified.

## Compile & Run Instructions

**Software:** Visual Studio is required to run this project.

**Additional Packages:** Xamarin must be installed with Visual Studio for the project to run.

- Choose one of the three versions of the project to run (base sequential, best sequential, or parallel).
- Open the project with Visual Studio using the 'DigitalMusicAnalysis.sln' file.
- Run the DigitalMusicAnalysis application.
- When prompted for a WAV file use the 'Jupiter.wav' file in the 'test-music' folder.
- When prompted for an XML file use the 'Jupiter.xml' file in the 'test-music' folder.

## Conclusion

In conclusion a significant speedup was achieved with the overall runtime of the best sequential program being cut from 2580.475ms down to 1077.463ms for the parallel version. I learnt a lot about parallelism and all the different factors that must be considered when analysing a program to make it run in parallel. While the speedup is significant, I believe there are still some improvements that could be made such as implementing a parallel version of the FFT algorithm like the Stockham FFT algorithm. This is an even more refined version of the Cooley – Tukey's FFT which I was unfortunately not able to implement due to time constraints. Overall, I am satisfied with the result and believe it was a successful project.

## References

Alyosify, M. (2023) Understanding and implementing the fast Fourier transform (FFT) algorithm in C#, Medium. Available at: <https://medium.com/@mahmoudalyosify/understanding-and-implementing-the-fast-fourier-transform-fft-algorithm-in-c-ff0a18cca7dc> (Accessed: 15 October 2023).

C# reversing the bits of an integer (2017) Stack Overflow. Available at: <https://stackoverflow.com/questions/47174754/c-sharp-reversing-the-bits-of-an-integer> (Accessed: 19 October 2023).

Fast fourier Transform (2022) Algorithms for Competitive Programming. Available at: <https://cp-algorithms.com/algebra/fft.html> (Accessed: 20 October 2023).

Kehtarnavaz, N. (2008) Short-Time Fourier Transform, Frequency Domain Processing. Available at: <https://www.sciencedirect.com/topics/engineering/short-time-fourier-transform> (Accessed: 21 October 2023).

What is an inverse fourier transform? (2023) Collimator. Available at: <https://www.collimator.ai/reference-guides/what-is-an-inverse-fourier-transform> (Accessed: 20 October 2023).

# Appendices

## Appendix 1

### Sequential

```

1 reference
float[][] stft(Complex[] x, int wSamp)
{
    int ii = 0;
    int jj = 0;
    int kk = 0;
    int ll = 0;
    int N = x.Length;
    float fftMax = 0;

    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[ll][ii] = (float)Complex.Abs(tempFFT[kk]);

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

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

    return Y;
}

```

## Parallel

```

1 reference
float[][] distributeSTFT(Complex[] x, int wSamp)
{
    N = x.Length;
    fftMax = 0;
    Y = new float[wSamp / 2][];
    xThread = x;

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

    Thread[] stftThreads = new Thread[MainWindow.availableThreadNum];
    numFramesPerThread = ((int)Math.Ceiling(2 * (double)N / (double)wSamp) + MainWindow.availableThreadNum) / (MainWindow.availableThreadNum);

    for (int j = 0; j < stftThreads.Length; j++)
    {
        stftThreads[j] = new Thread(stft);
        stftThreads[j].Start(j);
    }

    for (int j = 0; j < stftThreads.Length; j++)
    {
        stftThreads[j].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;
}

```

```

1 reference
public void stft(object? threadId)
{
    int id = (int)threadId;
    int startIndex = id * numFramesPerThread;
    int endIndex = startIndex + numFramesPerThread;
    if (endIndex > 2 * Math.Floor((double)N / (double)wSamp) - 1)
    {
        endIndex = 2 * (int)Math.Floor((double)N / (double)wSamp) - 1;
    }

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

    for (int k = startIndex; k < endIndex; k++)
    {
        for (int j = 0; j < wSamp; j++)
        {
            temp[j] = xThread[k * (wSamp / 2) + j];
        }

        tempFFT = FourierTransform.fft(temp, twiddles);

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

            if (Y[j][k] > fftMax)
            {
                fftMax = Y[j][k];
            }
        }
    }
}

```

## Appendix 2

```

0 references
public MainWindow()
{
    availableThreadNum = Environment.ProcessorCount * 2;
    parallelOptions = new ParallelOptions { MaxDegreeOfParallelism = availableThreadNum };

    Stopwatch sw = new Stopwatch();
    double timeElapsed;

    // Async Parallel
    sw.Start();
    InitializeComponent();
    sw.Stop();
    filename = openFile("Select Audio (wav) file");
    string xmlfile = "";
    Parallel.Invoke(
        () =>
        {
            xmlfile = openFile("Select Score (xml) file");
        },
        () =>
        {
            sw.Start();
            loadWave(filename);
            freqDomain();
            sw.Stop();
        }
    );
    sw.Start();
    Thread check = new Thread(new ThreadStart(updateSlider));
    sheetmusic = readXML(xmlfile);
    onsetDetection();
    loadImage();
    loadHistogram();
    playBack();
    check.Start();

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

    slider1.ValueChanged += updateHistogram;
    playback.PlaybackStopped += closeMusic;

    sw.Stop();
    timeElapsed = sw.Elapsed.TotalMilliseconds;
    Debug.WriteLine("Time elapsed until main winow opened = {0}ms", timeElapsed);
}

```