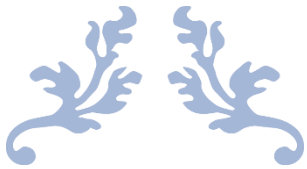


CAB401 PARALLELISATION

NICHOLAS MEURANT N10485571

OCTOBER 28, 2022

QUEENSLAND UNIVERSITY OF TECHNOLOGY

Contents

Table of Figures

Explanation of Original Application

The chosen application is the music analyser. The purpose of the music analyser is to enable users to verify the correctness of their recorded music against a music sheet. This is especially useful for users who do not have direct access to musical teachers. The music analyser works by converting the original sound file to the frequency domain and comparing the frequency of the recorded sound with the expected frequency stated on the music sheet.

Class Explanation

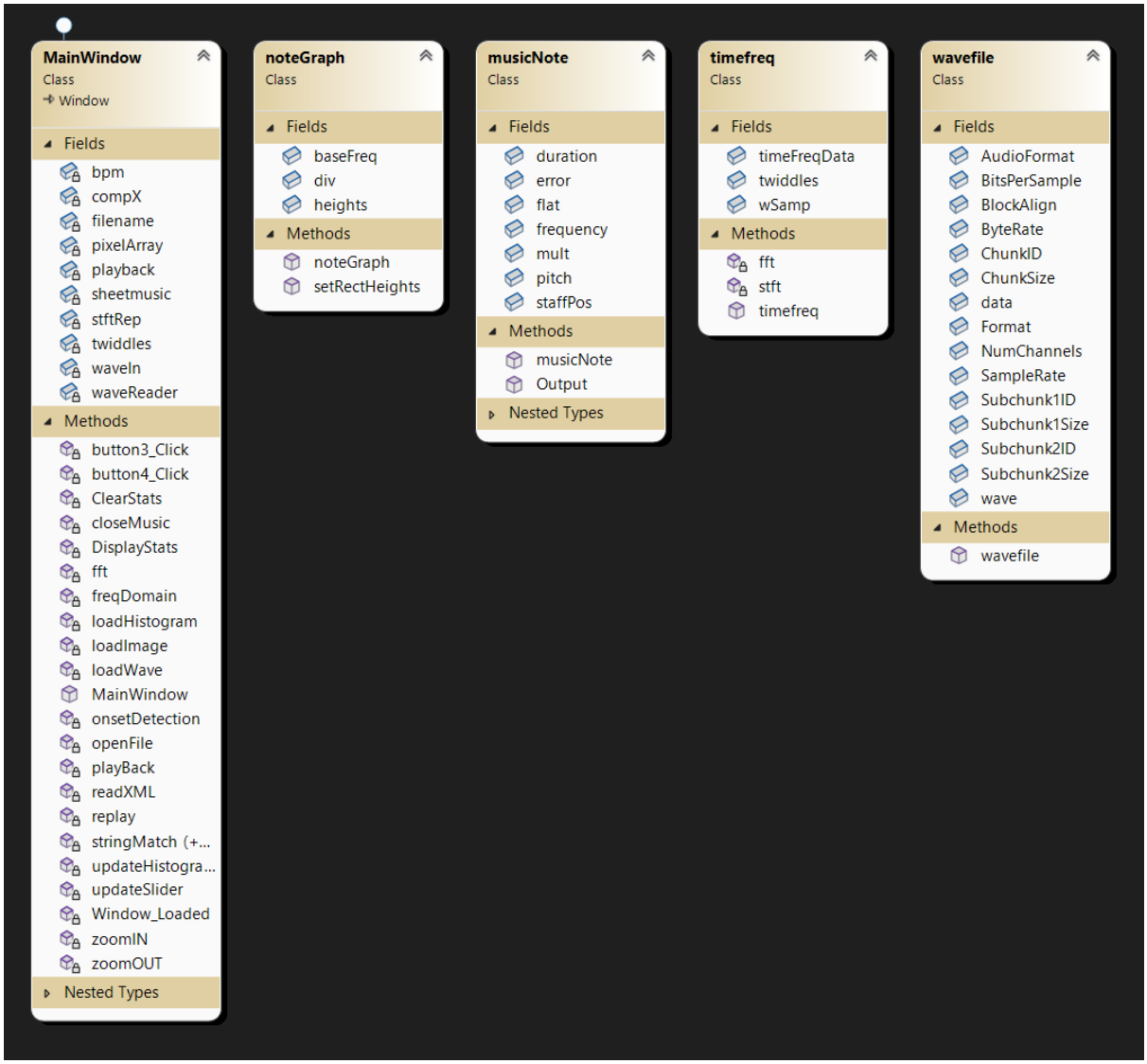


Figure 1 Class Diagram

Figure 1 refers to the class diagram representation of the music analyser. Five major classes exist for the program. These 5 classes are, MainWindow, noteGraph, musicNote, timefreq, and waveFile. Of these 5 classes noteGraph, musicNote and waveFile are all "helper" functions that are there to display the results of the analysis. Most of the computation is found in the MainWindow and the timefreq classes.

MainWindow

This class acts as the main function of the application. The MainWindow class contains code that sequentially calls all the other classes to do the music analysis. This class contains the code that enables the user to select their music sheet and their recording. This class contains a wide selection of different methods, the most

important of these methods are fft and onsetDetection. The fft stands for “fast Fourier transform” and it is used for converting the sound from the time domain to the frequency domain. The onsetDetection method calls the fft method to convert from time to the frequency domain, it then compares the expected frequency of sound against the recorded sound and returns a rating of accuracy. The onsetDetection method is a significant source of computation with it taking 35-40% of the total computation time.

noteGraph

noteGraph is an object class that is used to visually display the analysis on a graph. This class uses little computation and will not be considered for parallelism.

musicNote

musicNote is another object class that is used to identify a musical note from the frequency and duration of the note played. This class is called when the user inputs the .wav file, the results from this class are then used when comparing the recording and the music sheet.

timeFreq

timeFreq is where a lot of the sound analysis is contained. This class contains the short-time Fourier transform (STFT) and a complex fft method. The STFT function is used over the traditional fft function 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” (Kehtarnavaz, 2008). The STFT function provides higher accuracy because it is taking a linear approximation over small intervals. As a result, the STFT function takes a significant amount of computation and is a section of code worth exploring for potential parallelism.

waveFile

This is another object class that is used to read the user’s recording and for it to be used for analysis further in the application.

Call Graph

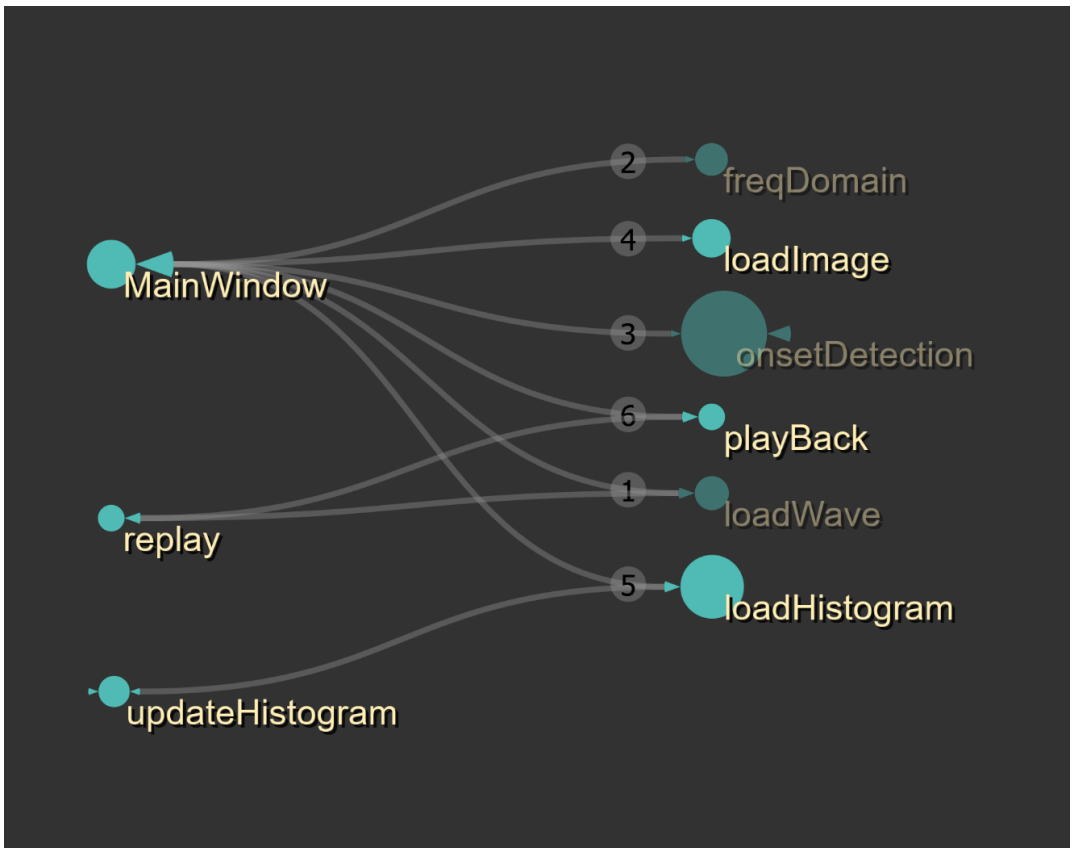


Figure 2 Music Analyser Call Graph

Figure 2 refers to the method calling that is present in the applications lifecycle, there are numbers on the figure to illustrate the sequence of the calls. First, the MainWindow is called because it is the entry point of the application, the MainWindow calls the loadWave method to load in the sound wave of the user. After loading in the sound wave, the application calls the freqDomain method which in turn calls the timefreq class mentioned previously to compute the STFT. After calling the freqDomain method, the onsetDetection method is called which analyses the accuracy of the notes in comparison to the music sheet. After the onset detection is called, the loadImage method is called, this method loads an image that visually displays the accuracy of the user's recording. After the loadImage method is called, the loadHistogram method is called, this method displays a histogram of the accuracy of the user's notes. All the other calls made after this are to allow the user to replay the music.

Analysis of Potential Parallelism

To find methods that are most appropriate to speed up, benchmarks of each method will need to be taken. C# by default has an object called "stopwatch" that can be used to take the time of execution. The methods that will be evaluated are the freqDomain, onsetDetection, STFT method, and a time that records the total computation of the application (appendix 1 and 2). It is not worth to analyse any methods that are to do with displaying information to the user because these methods do not seem to take a lot of computation to complete.

Section of Code	Time Taken (mS)
Total Time	3833
Freq Domain	2100
onsetDetection	1637
STFT	2041

Table 1 StopWatch benchmark

Table 1 refers to the time taken for each section. The total time of computation for all sections except the visual display of results is 3833. The freqDomain section of the code took approximately 500ms longer than the onsetDetection section of the code. As mentioned previously, the freqDomain method calls the timeFreq method which then runs the STFT for the music wave. It was found that the STFT method was the major computational bottleneck associated with the freqDomain and therefore will be the section of code most closely explored for potential speedup.

freqDomain Analysis

Due to the results found in table 1, it is found that significant computation is present and therefore a lot of potential for speedup. According to table 1, a large majority of the time spent is within the STFT function so therefore, this method will be explored the most in-depth. On line 72 (appendix 3) there appears to be a for loop that does not contain any data dependencies and could potentially be run in parallel. However, the computation does not appear to be very intensive, and the overhead of spawning additional threads might be greater than the parallel speedup. On line 80 (appendix 3), there is a for loop that calls the FFT on small sections of the sound wave. When testing for "Jupiter.wav" the loop condition runs 1164 times, this is a number large enough that parallelism would be of significant benefit. Inside the loop, there appears to be no data dependency that gets carried through each loop and the identification of FFTmax can be found later. When looking at the FFT function (appendix 4), the FFT function is recursive and appears to be slow. When tracking the total number of times this FFT function was called, it was found that it had been called 9529065 times in total for the "Jupiter.wav" file. This is a significant amount of computation where the overhead of spawning additional threads would be less than the time saved through parallelism and therefore the loop on line 80 will be parallelised. Furthermore, it may be beneficial to also investigate modifying the FFT function itself to another faster FFT (assuming the output can be kept the same).

onsetDetection Analysis

In table 1 it was found that the onsetDetection took a significant amount of time and therefore was worth investigation for potential parallelism. Like the STFT function, the onsetDetection method contains a for loop that calls a complex FFT method on lines 390 and 415 (appendix 5 and 6) respectively. The for loop on line 390 seems like a great candidate for parallelism because there are no data dependencies on each loop and can safely be run in parallel. One potential issue is that when running the "Jupiter.wav" file, the loop only executes seventy-two times, this could potentially reduce the effectiveness of parallelism when using a higher thread count (12+). The FFT function that is present in the onsetDetection method is still a recursive FFT but takes in an additional input parameter of length, which will therefore modify the twiddles array causing a rewrite to be of significant difficulty.

Given the fact that 99% of the computational time is found between these two previously mentioned methods, the speedup should be quite significant and is therefore worth exploring.

Parallelism Methodology

In the sections previous it was identified that the STFT function and the onsetDetection functions were the most computationally intensive and thus worth exploring parallelism. The STFT function will be parallelised by replacing the for loop located on line 80 (appendix 3) with a Parallel. For loop that is present in the C# language. When making this change, errors started to appear because multiple threads were trying to access the same variable and each thread was overwriting the variable. To overcome this issue, the Parallel .For loop reinitialised the variable locally to ensure that each thread was given its own "version" of the variable. The last issue to overcome was to change the logic of how the fftmax variable was generated. In the sequential version of the code, the fftmax was calculated during the loop however, this same logic won't work for the parallel version. To overcome this issue, an array was initialised outside of the Parallel.For loop where each thread would append their respective fft max for each loop they completed. After the completion of the Parallel.For loop, this array will then be iterated through to find the true fftmax. The rest of the STFT will remain the same.

The onsetDetection will be handled in a similar manner. It was identified that the for loop on line 390 (appendix 5) was computationally intensive and could be safely ran in parallel. The for loop on line 390 was replaced with a Parallel.For loop. This change ran into similar issues as the STFT method where multiple threads were writing to the same variable and overwriting previous values. To overcome this issue, the local variables were reinitialised in each Parallel.For to ensure that each thread had their own "version" of the variable. One further issue was that when the threads were writing to the pitches list, this was being done at random intervals and was causing issues further down in the onsetDetection method. To overcome this issue, separate pitches arrays were initialised for each Parallel.For thread. After the pitches were added to their individual arrays, the arrays were added back to replicate the output of the previous sequential code.

FFT Modifications

During testing it was obvious that the recursive fft was a major bottleneck for performance. After researching various fft .NET implementations it became clear that the Fastest Fourier Transform in the West (FFTW) (Woltering, 2022). The FFTW (Matteo Frigo, 2022) is a FFT algorithm developed by Steven G. Johnson and Matteo Frigo, it is commonly benchmarked as the fastest FFT implementation possible. It is originally programmed in C but it's DLLs are made publicly available for use. For this assignment, the FFT method that is present in the freqDomain class can be replaced with the FFTW implementation while keeping the output the same. The FFT function present in the onsetDetection method will have to be kept the same because the method is slightly different and will require a significant rewrite that is out of the scope of this assignment. When converting the original recursive FFT to the FFTTW it is hoped that reducing the number of calls to the call stack will drastically improve performance when parallelising the FFT calls. Implementation of the FFTW is quite simple due to an existing nuGet package that directly calls the FFTW DLLS, screen shots will be provided later.

Results

To attain timing results various stopwatches were used throughout the application. There were three stopwatches used in total, one stopwatch around the entire application called "TotalTime", one stopwatch around freqDomain called "freqDomain" and the last stopwatch was wrapped around onsetDetection called "onsetDetection". After every time the application was ran, these values were passed to a "benchmark.txt" file and that data was passed into excel to find the average. Every test was done a total of 10 times and their average was recorded. All results were recorded while the application was in "release mode"

Sequential Program Benchmark

TotalTime Average (ms)	FreqDomain Average (ms)	onsetDetection Average (ms)
2830.7	1501.7	1312.8

Table 2 Sequential Program Result Table

Table 2 refers to the benchmark of the sequential program. There appears that the freqDomain and onsetDetection methods take approximately the same amount of time with freqDomain only taking slightly longer.

Parallelised Program Results

Like the sequential program, the parallel program used the same stopwatches and tested each number of cores 10 times, the results were then written to a "benchmark.txt" file. However, the benchmark file now also contained the total number of threads used for parallelism as well. A python script was also created to continuously run the built executable file.

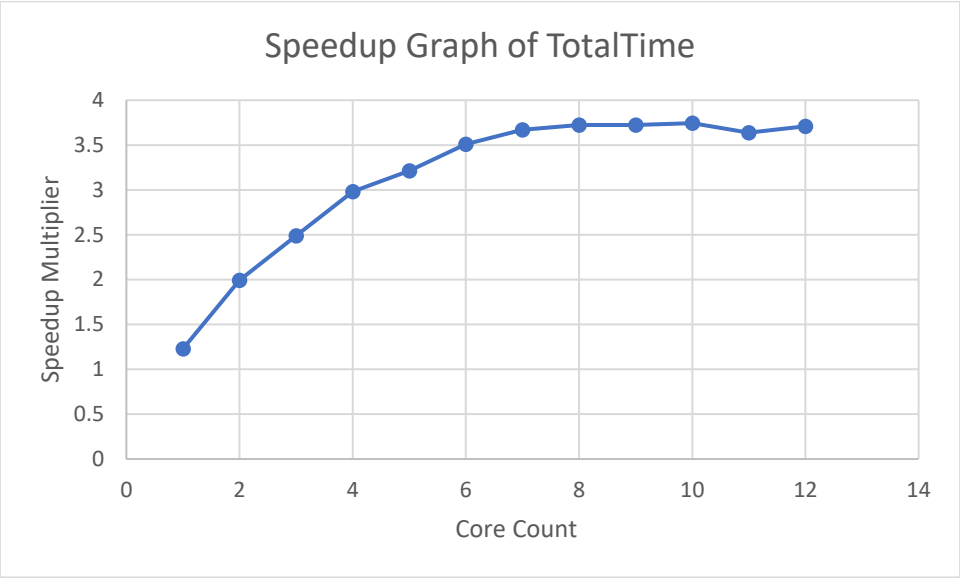


Figure 3 TotalTime Speedup Graph

Figure 3 refers to the speedup of the TotalTime stopwatch appendix (7) shows a tabulated form of this data with appendix 8 and 9 showing a graphical representation of freqDomain and onsetDetection.

From the graph it appears at that the speedup is approximately linear from 1 – 6 core count with the speedup reaching diminishing returns after more than 6 cores. The fastest performance experienced for the parallel program was at 10 core count with a totalTime speedup of 3.74x. When looking at the timeFreq and the onsetDetection, the onsetDetection is not benefitting from parallelism as much as the freqDomain has. One potential reason for this is because the FFT method in onsetDetection is still the recursive version of the code. Due to the recursive nature of the FFT millions of calls are made to the same method between multiple threads causing the memory manager to have high overhead to ensure memory/thread safe execution. The last reason is that the for loop present on line 390 (appendix 5) would only execute 72 times. That means that the overhead of spawning additional threads was greater than the performance gain at higher core counts. The updated FFT method present in the freqDomain experienced nearly a 10x speedup when using all 12 cores. One explanation for this is that the FFTW DDLS are optimised for parallelism and don't require as much overhead when being called by multiple threads. The onsetDetection method saw the best results using 6 cores (speedup of 2.38) and saw decreased performance for more cores.

From observing figure 3, the speedup experienced using 1 core is ~1.23, it should be expected that single core performance matches the performance of the sequential. The reasoning for this not occurring for the parallel code is because the FTTW implementation of the STFT is significantly faster than the original sequential code event at one core.

Profiling Results

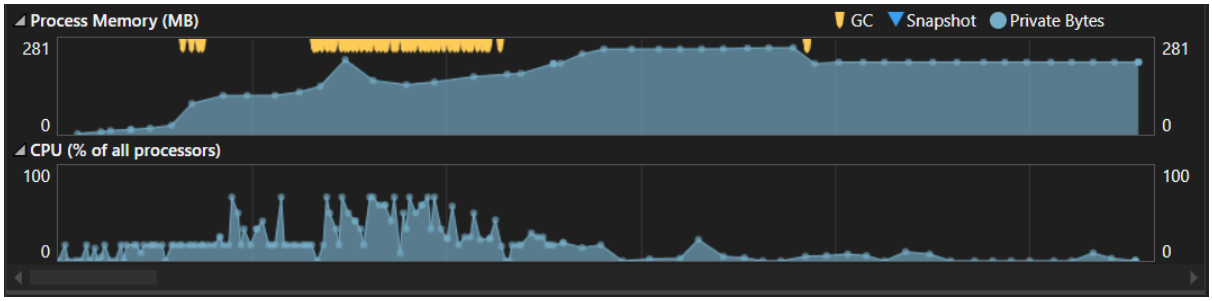
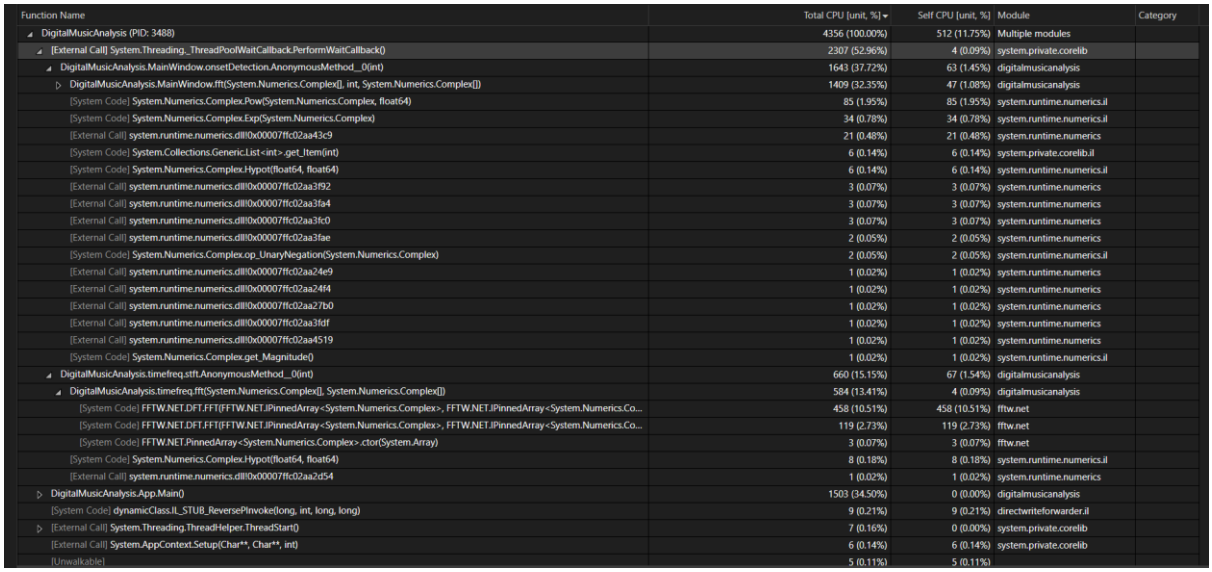


Figure 4 Memory and CPU Utilisation

Figure 4 refers to the memory and CPU utilisation of the parallel program. The memory utilisation very quickly increases, this indicates that the additional threads are utilising memory. The CPU spikes very quickly as well, these spikes most likely correlate to sections of code where the additional threads are spawned. The latter half of the CPU utilisation graph is the section where the music is playing and thus uses very little processing.



Function Name	Total CPU [unit, %]	Self CPU [unit, %]	Module	Category
DigitalMusicAnalysis (PID: 3488)	4356 (100.00%)	512 (11.75%)	Multiple modules	
[External Call] System.Threading._ThreadPoolWaitCallback.PerformWaitCallback()	2307 (52.96%)	4 (0.09%)	system.private.corelib	
DigitalMusicAnalysis.MainWindow.onsetDetection.AnonymousMethod__0(int)	1643 (37.72%)	63 (1.45%)	digitalmusicanalysis	
DigitalMusicAnalysis.MainWindow.fft(System.Numerics.Complex[], int, System.Numerics.Complex[])	1409 (32.35%)	47 (1.08%)	digitalmusicanalysis	
[System Code] System.Numerics.Complex.Pow(System.Numerics.Complex, float64)	85 (1.95%)	85 (1.95%)	system.runtime.numerics.il	
[System Code] System.Numerics.Complex.Exp(System.Numerics.Complex)	34 (0.78%)	34 (0.78%)	system.runtime.numerics.il	
[External Call] system.runtime.numerics.dll!0x00007ffc02aa43c9	21 (0.48%)	21 (0.48%)	system.runtime.numerics	
[System Code] System.Collections.Generic.List<int>.get_Item(int)	6 (0.14%)	6 (0.14%)	system.private.corelib.il	
[System Code] System.Numerics.Complex.Hypot(float64, float64)	6 (0.14%)	6 (0.14%)	system.runtime.numerics.il	
[External Call] system.runtime.numerics.dll!0x00007ffc02aa3f92	3 (0.07%)	3 (0.07%)	system.runtime.numerics	
[External Call] system.runtime.numerics.dll!0x00007ffc02aa3fa4	3 (0.07%)	3 (0.07%)	system.runtime.numerics	
[External Call] system.runtime.numerics.dll!0x00007ffc02aa3fc0	3 (0.07%)	3 (0.07%)	system.runtime.numerics	
[External Call] system.runtime.numerics.dll!0x00007ffc02aa3fae	2 (0.05%)	2 (0.05%)	system.runtime.numerics	
[System Code] System.Numerics.Complex.op_UnaryNegation(System.Numerics.Complex)	2 (0.05%)	2 (0.05%)	system.runtime.numerics.il	
[External Call] system.runtime.numerics.dll!0x00007ffc02aa24e9	1 (0.02%)	1 (0.02%)	system.runtime.numerics	
[External Call] system.runtime.numerics.dll!0x00007ffc02aa24f4	1 (0.02%)	1 (0.02%)	system.runtime.numerics	
[External Call] system.runtime.numerics.dll!0x00007ffc02aa27b0	1 (0.02%)	1 (0.02%)	system.runtime.numerics	
[External Call] system.runtime.numerics.dll!0x00007ffc02aa3fdf	1 (0.02%)	1 (0.02%)	system.runtime.numerics	
[External Call] system.runtime.numerics.dll!0x00007ffc02aa4519	1 (0.02%)	1 (0.02%)	system.runtime.numerics	
[System Code] System.Numerics.Complex.get_Magnitude()	1 (0.02%)	1 (0.02%)	system.runtime.numerics.il	
DigitalMusicAnalysis.timefreq.stft.AnonymousMethod__0(int)	660 (15.15%)	67 (1.54%)	digitalmusicanalysis	
DigitalMusicAnalysis.timefreq.fft(System.Numerics.Complex[], System.Numerics.Complex[])	584 (13.41%)	4 (0.09%)	digitalmusicanalysis	
[System Code] FFTW.NET.DFT.FFT(FFTW.NET.IPinnedArray<System.Numerics.Complex>, FFTW.NET.IPinnedArray<System.Numerics.Co...	458 (10.51%)	458 (10.51%)	fftw.net	
[System Code] FFTW.NET.DFT.FFT(FFTW.NET.IPinnedArray<System.Numerics.Complex>, FFTW.NET.IPinnedArray<System.Numerics.Co...	119 (2.73%)	119 (2.73%)	fftw.net	
[System Code] FFTW.NET.PinnedArray<System.Numerics.Complex>.ctor(System.Array)	3 (0.07%)	3 (0.07%)	fftw.net	
[System Code] System.Numerics.Complex.Hypot(float64, float64)	8 (0.18%)	8 (0.18%)	system.runtime.numerics.il	
[External Call] system.runtime.numerics.dll!0x00007ffc02aa2d54	1 (0.02%)	1 (0.02%)	system.runtime.numerics	
DigitalMusicAnalysis.App.Main()	1503 (34.50%)	0 (0.00%)	digitalmusicanalysis	
[System Code] dynamicClass.IL_STUB_ReversePInvoke(long, int, long, long)	9 (0.21%)	9 (0.21%)	directwriteforwarder.il	
[External Call] System.Threading.ThreadHelper.ThreadStart()	7 (0.16%)	0 (0.00%)	system.private.corelib	
[External Call] System.AppContext.Setup(Char**, Char**, int)	6 (0.14%)	6 (0.14%)	system.private.corelib	
[Unwalkable]	5 (0.11%)	5 (0.11%)		

Figure 5 Parallel Program Call Tree

Figure 5 refers to the call tree of the parallel program. Figure 5 shows the CPU utilisation of each of the various methods, it appears that over 50% of the CPU was being consumed by the thread pool call-back method. Unsurprisingly, the unoptimised recursive FFT consumed over 33% of the total CPU due to its recursive nature. The FFTW implementation the FFT took consumed only 13.41% of the total CPU. Consequentially, the FFTW implementation and is both significantly faster and is less intensive on the CPU.

Amdahl's Law

Total Time	FreqDomain (Parallel)	onsetDetection (Parallel)	Parallel Percentage
3788	1955	1632	0.9469
3888	2053	1643	0.9506
3902	2077	1631	0.9503
3852	2049	1619	0.9522
3861	2035	1632	0.9498

Table 3 Table of Parallel Percentage

$$Speedup = \frac{1}{(1-P) + (\frac{P}{N})}$$

Where P = percentage of parallel time, N = total number of threads

Since the benchmarks were performed on a dual core 6 core processor, the total number of threads is 12. From table 1, P = 0.95 and S = 0.05

$$Speedup = \frac{1}{(1-0.95) + (\frac{0.95}{12})}$$

Speedup = 7.742.

From the equation it can be said that the maximum potential speedup achievable is 7.742. This is obviously lower than the 3.74 times speedup achieved through the parallel speedup. One potential reason for this is

because the recursive FFT function still exists within the onSetDetection method, and that section is not able to utilise the extra threads as effectively. Overall, however, Amdahl's law describes the upper limit and is nearly impossible to be achieved. This report believes a speedup 3.74 is still extremely good.

Testing Methodology

To make sure that the results from the sequential and parallel versions of the code are the same, the values of the pitch array were written to a text file. These text files were then compared to one another using a small python script (appendix 10). Appendix 10 shows the python script, essentially the script compares line by line of each text file and prints the difference to the console. After printing the result, it was found that the calculated pitches were the same.

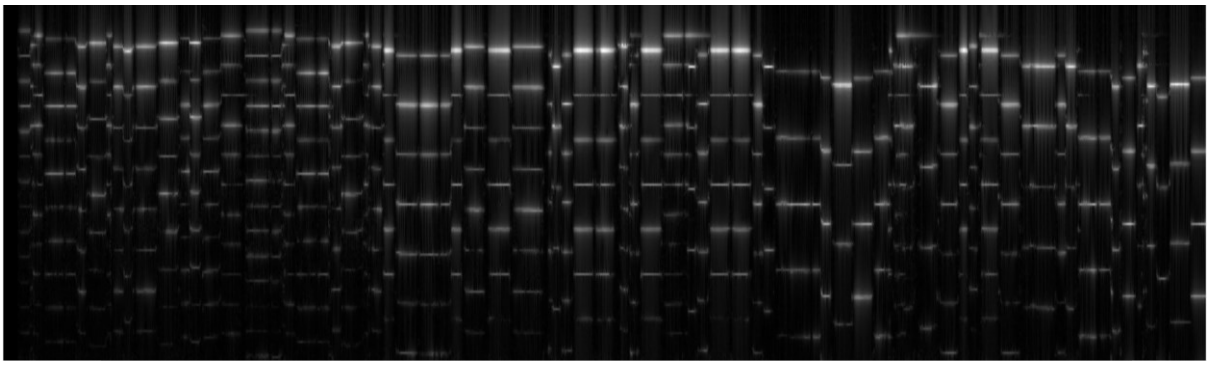


Figure 6 Parallel Graphical Output

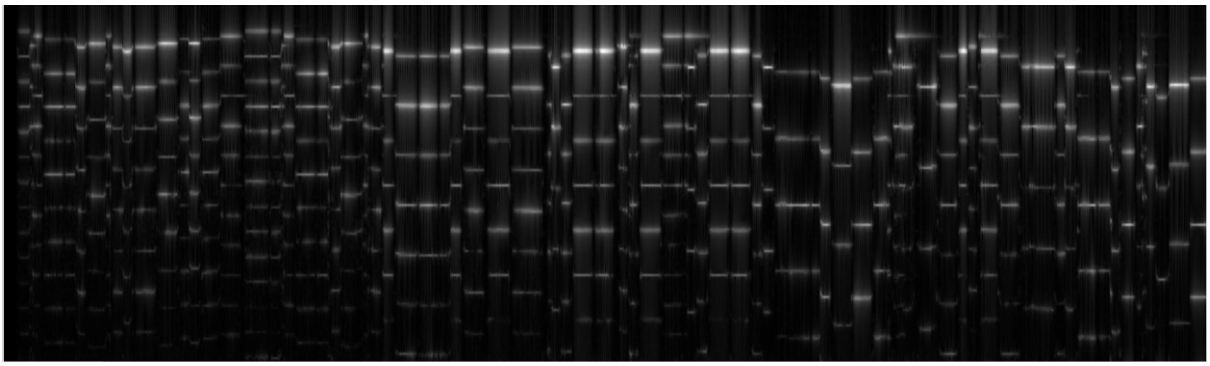


Figure 7 Sequential Graphical Output

Another method to spot any differences is to compare the graphical output of the two different programs. Figures 5 and 6 refer to the parallel and sequential graphical output respectively. The graphs show the note pitch and noise, with respect to time. When comparing both figures, it appears that the outputted pitch, noise, and timing are the same.

The last method that can be used to verify the output is to compare the staff output of both programs.

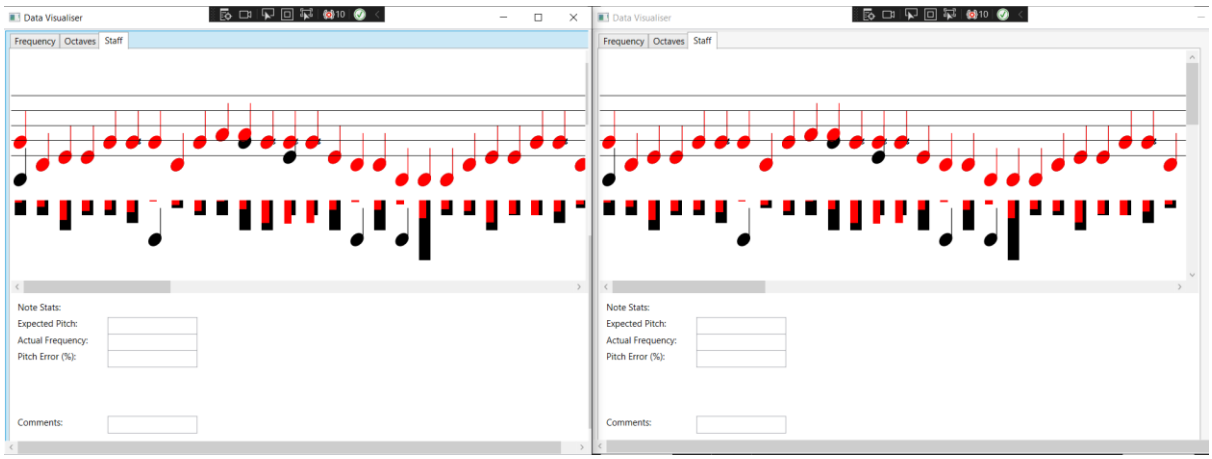


Figure 8 Staff Output Comparison (sequential left, parallel right)

Figure 7 refers to the staff output of the sequential and parallel program (left to right respectively). By visually looking it appears that both applications outputting the same pitch errors. Furthermore, when "hovering" over each note, both graphs have the same percentage pitch error.

After passing all three of these difference checks, it can be said with high certainty that the sequential and parallel versions of the code produce the same values.

Description of Software Tools used

System.IO

System.IO was used to allow the use of stream writer and text writer. Stream writer and text writer were needed because it allowed the program to write to the benchmark.txt and output.txt to help with benchmarking and verification of correct output.

Visual Studio 2022

Visual Studio 2022 was the IDE of choice for this project. The reasoning for choosing Visual Studio 2022 is due to its large amount of different support and its easy-to-use NuGet package manager. The NuGet Package manager was used to download the FFTW files.

FFTW.NET

FFTW.NET is another software tool that was used to implement the FFTW function that resulted in the extreme performance increase at higher core counts. Without the use of this tool the speedup multiplier would follow a similar curve as the onSetDetection graph and achieved a far lower overall speedup.

System.Threading

System.Threading was a crucial tool that was used throughout the duration of this project. System.Threading is needed to spawn additional threads and is necessary to implement the Parallel.For loops that were used for the speedup. Without the use of this software tool, it would be impossible to achieve a speedup of 3.7.

Overcoming Challenges

There were a few challenges that had to be overcame to create the successful application. The largest of the issues had to have been correctly identifying data dependencies in the program so that effective parallelism could be achieved. As I have progressed through this unit, I have become more confident with parallelising software. While attempting this project early in the semester I found it very unintuitive to analyse software data dependencies. As the semester progressed and I practiced, it became very easy to identify these dependencies. As a result, I was able to analyse the sequential code and identify the two major for-loops for which parallelism was possible quite quickly. Furthermore, it also took quite a lot of time to understand that

each thread has its own “copy” of a variable and thus you must redeclare each variable when running code in parallel.

Another issue that I had to overcome was refactoring the code to use arrays/lists to store variables that otherwise would have just been updated sequentially. An example of this is the FFTMax array that exists in the STFT function in timefreq.cs. It took quite a lot of time to realise that the local value of each thread’s loop could safely be placed into its respective index in an array and to then find the max after the loop had been executed completely.

The last issue to overcome was to realise that the speedup of the FFTW comes from importing and exporting a file commonly named “wisdom.txt”. Wisdom.txt files “contain saved information about how to optimally compute (Fourier) transforms of various sizes” (Steven G. Johnson, 2003). Before using the wisdom.txt file, performance was comparable to the existing recursive code however, it sped up significantly after importing/exporting the wisdom.txt files.

Explanation of Modified Code

As mentioned previously, onSetDetection and timefreqs STFT functions have both been modified. The following section will highlight the changes made to the assignment.

Time Freq STFT

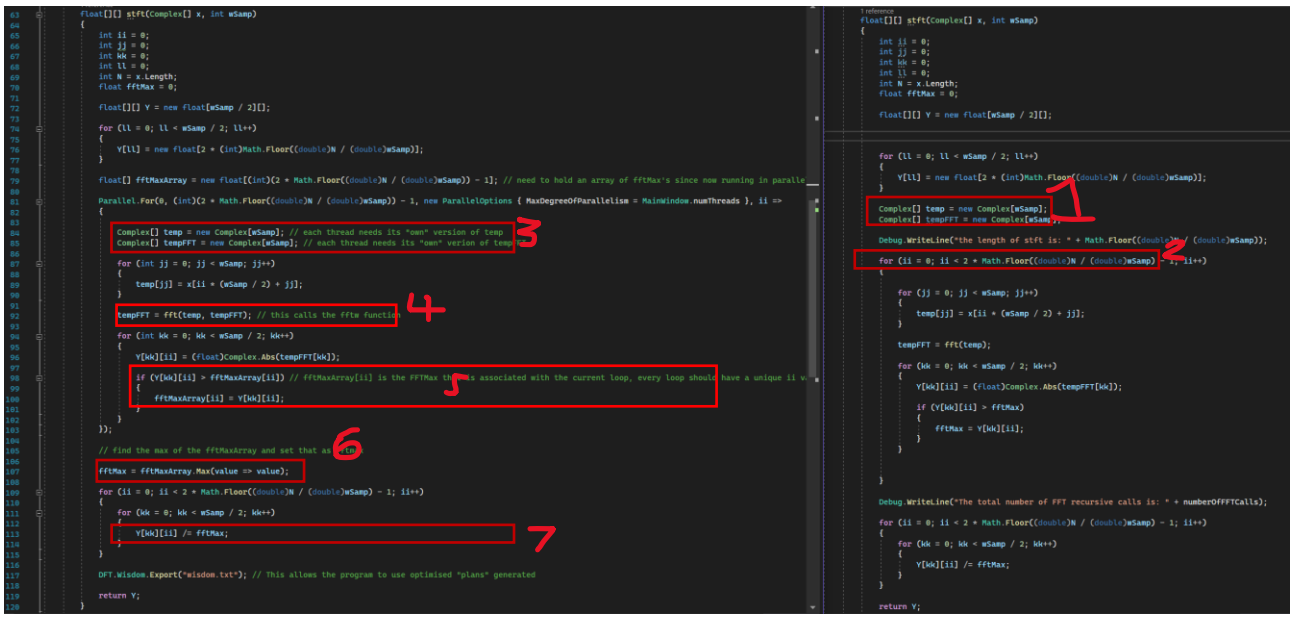


Figure 9 TimeFreq STFT Function Difference (Parallel Left, Sequential Right)

1. The first change was to make the temp and tempfft arrays to no longer be global since multiple threads would be accessing them.
2. The second change was to change the sequential for loop to a Parallel.For loop.
3. The third change was to add back the temp and tempfft arrays to local scope of the for loop
4. The fourth change has to do with the FFT function. As mentioned previously, this was changed to the FFTW implementation (appendix 11).
5. The fifth change the fftmax value to be an array instead of an individual number. The reasoning for this change is because each thread will not have a current “global” value of fftmax. To avoid this sharing of the fftmax value, an array was implemented to ensure that each thread would simply place their computed value into a specific index in the array.
6. Since fftmax is now an array, a line of code was added to find the fttmax from the fftmax array. This change has been verified to give the same result as the sequential implementation.

7. The last change was to export a file called “wisdom.txt”. Essentially this file will store the computed values from the FFTW and store them in a file to avoid computing them again. Importing/exporting the “wisdom.txt” file significantly reduces computation time.

OnsetDetection

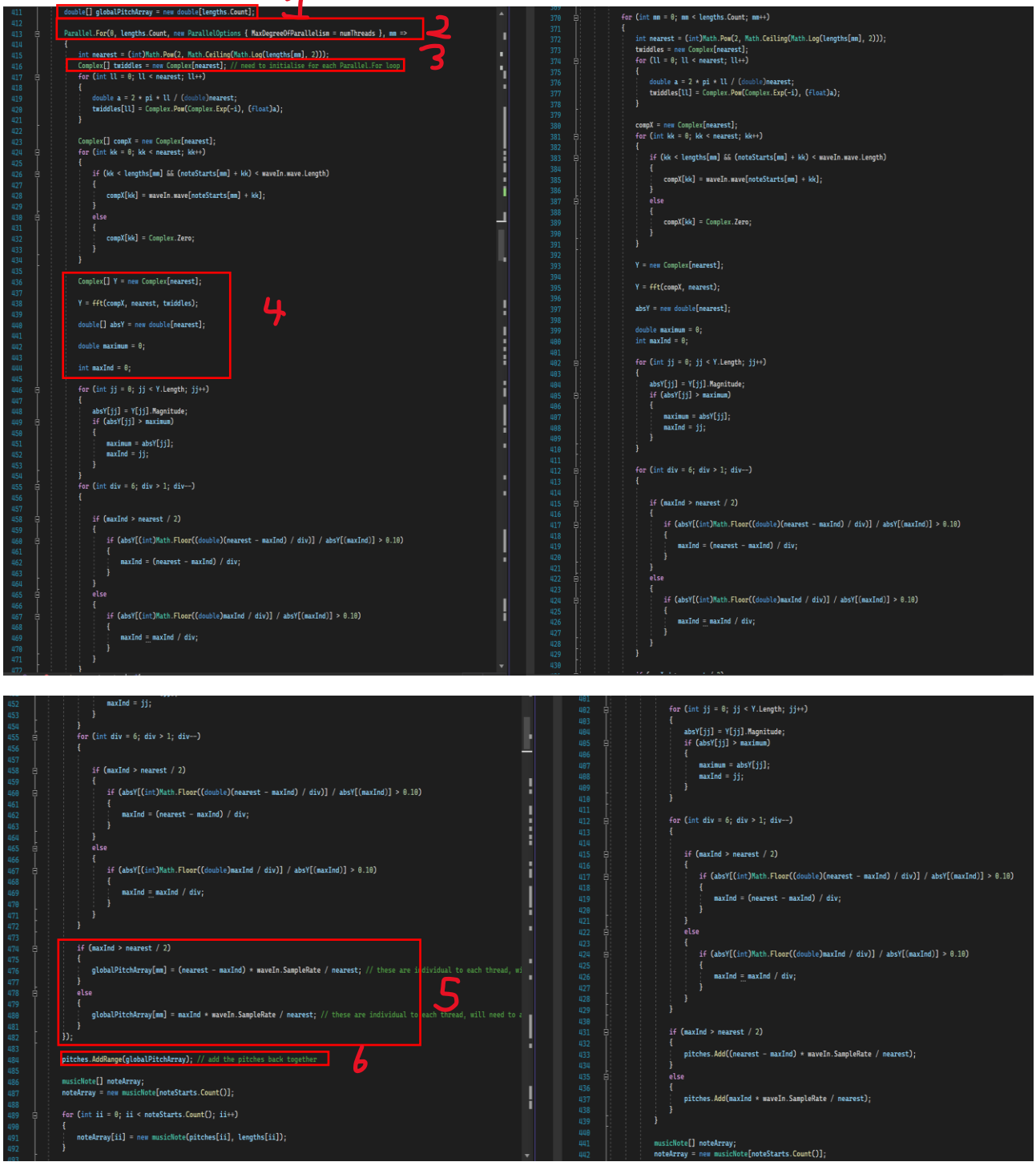


Figure 10 OnsetDetection Changes (Parallel Left, Sequential Right)

1. The first change was to create a global list that contains the pitches. The reasoning for this is to allow each thread to push their result to an index within the array, this ensures that each thread is pushing to a different index in the array.
2. The second change is to implement a Parallel.For loop instead of a sequential for loop.
3. The third change is to create a local twiddles array for each loop of the Parallel.For loop. The reasoning for this is because the sequential version referenced a global version of the twiddle array.

When the code was running sequentially, it was fine to use the global version because it would only be changed on every loop. Now that the code is parallel, multiple threads will be editing the global value and this will cause errors during calculations. To remedy this issue, a local version of the twiddles array will need to be used for each Parallel.For loop.

4. The fourth change is to change the Y value to be local to the loop, this again is to stop each of the threads editing the same global variable. The FFT has edited to now include a twiddles array parameter. The reasoning for this is because the previous version of the FFT was using a global twiddles array. However, now that the program is running in parallel, each thread/loop needs its own local version of the twiddles array. The rest of the FFT function is largely unchanged and is still using the recursive algorithm due to refactoring being out of the scope of this unit. Lastly, there is now a local version of the absY value for the same reason as the Y value mentioned previously.
5. The fifth change is to now add each computed pitch to the index of the "globalPitchArray". The logic is the same, the reasoning for this change is due to if each thread is editing the global pitch, it would cause errors. Adding to an index within an array will ensure that each loop/thread, is sending information to different indexes within the array.
6. The last change is to add all the local pitch values back into the global pitch array. Adding the local pitches at this stage ensure that the remainder of the onSetDetection function will operate in the exact same manner as previous.

STFT FFT

```
1 reference
Complex[] fft(Complex[] temp, Complex[] tempFFT)  // uses FFTW
{

    using (var pinIn = new PinnedArray<Complex>(temp))
    using (var pinOut = new PinnedArray<Complex>(tempFFT))
    {
        DFT.FFT(pinIn, pinOut); // computes the fft using the FFTW algorith
    }

    return tempFFT;
}
}
```

```
}

3 references
Complex[] fft(Complex[] x)
{
    numberOfFFTCalls++;
    int ii = 0;
    int kk = 0;
    int N = x.Length;

    Complex[] Y = new Complex[N];

    if (N == 1)
    {
        Y[0] = x[0];
    }
    else{

        Complex[] E = new Complex[N/2];
        Complex[] O = new Complex[N/2];
        Complex[] even = new Complex[N/2];
        Complex[] odd = new Complex[N/2];

        for (ii = 0; ii < N; ii++)
        {

            if (ii % 2 == 0)
            {
                even[ii / 2] = x[ii];
            }
            if (ii % 2 == 1)
            {
                odd[(ii - 1) / 2] = x[ii];
            }
        }

        E = fft(even);
        O = fft(odd);

        for (kk = 0; kk < N; kk++)
        {
            Y[kk] = E[(kk % (N / 2))] + O[(kk % (N / 2))] * twiddles[kk * mSamp / N];
        }
    }

   return Y;
}
}
}
```

Figure 11 STFT FFT Modification (Parallel Left, Sequential Right)

Figure 11 refers to the FFTW implementation of the FFT. As mentioned previously, FFTW is a library that is originally written in C and thus needs to be called through exporting the code as DLL's. There is a convenient C# wrapper that is downloaded as a NuGet package (ArgusMagnus, 2017). It was quite trivial to implement the FFTW implementation due to the STFT FFT using a length that is related to sampling rate. This is in contrary to the OnsetDetection FFT which uses a length that is determined per loop.

1. Removed the recursive code and replaced it with a C# wrapper that will call the C DLL's
2. Removed the need to create a new Complex array store the array. This change was achieved by passing the array into the function as a parameter and returning it. The code to create the array can be seen in figure 9.

Other than writing values to a benchmark.txt or output.txt file and stopwatches, the remainder of the code base has not been modified and is operating identically.

Reflection

I believe that the speedup for this project has been a major success. The reasoning for this success is because the application saw a 3.7x speedup when compared to the original sequential code. Furthermore, this speedup was achieved while keeping the output of the application the same.

One issue with this project, however, is that the FFT in the onsetDection method is still the inefficient recursive version. The recursive version of the FFT function is a large bottle neck and only saw 2.38 times increase whereas the FFTW saw a near 10x improvement. If the OnsetDetection FFT was to be rewritten the overall project could have seen even greater speedup. I am, however, quite pleased with the result considering the large amount of work required to refactor the FFT function.

One change I would make when completing this assignment is to start the assignment after I have learnt parallel analysis thoroughly. I believe that I wasted a very significant amount of time trying to rewrite functions without a solid baseline understanding of parallelism.

Overall, I am very happy with the outcome of this assignment, and I believe a 3.7x speedup is a great result and this project has been a success.

Appendix

```
0 references
public MainWindow()
{
    InitializeComponent();
    filename = @"C:\Users\Nicho\OneDrive\Documents\CAB401\Assignment1\music\Jupiter.wav";
    string xmlfile = @"C:\Users\Nicho\OneDrive\Documents\CAB401\Assignment1\music\Jupiter.xml";
    Thread check = new Thread(new ThreadStart(updateSlider));

    Stopwatch totaTime = new Stopwatch();
    Stopwatch freqDomainTime = new Stopwatch();
    Stopwatch onsetDectionTime = new Stopwatch();

    totaTime.Start();

    loadWave(filename);

    freqDomainTime.Start();
    freqDomain();
    freqDomainTime.Stop();

    sheetmusic = readXML(xmlfile);

    onsetDectionTime.Start();
    onsetDetection();
    onsetDectionTime.Stop();

    totaTime.Stop();

    loadImage();
    loadHistogram();

    playBack();
    check.Start();

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

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

    Debug.WriteLine($"Total Time: {totaTime.ElapsedMilliseconds}\n" +
        $"Freq Domain Time: {freqDomainTime.ElapsedMilliseconds}\n" +
        $"Onset Detection Time: {onsetDectionTime.ElapsedMilliseconds}");
```

Appendix 1 stopWatch for freqDomain, onsetDetection and Total time

```
            {
                compX[kk] = x[kk];
            }
            else
            {
                compX[kk] = Complex.Zero;
            }
        }

        int cols = 2 * nearest /wSamp;

        for (int jj = 0; jj < wSamp / 2; jj++)
        {
            timeFreqData[jj] = new float[cols];
        }
        Stopwatch STFTStopWatch = new Stopwatch();
        STFTStopWatch.Start();
        timeFreqData = stft(compX, wSamp);
        STFTStopWatch.Stop();
        Debug.WriteLine($"STFT Time: {STFTStopWatch.ElapsedMilliseconds}");

    }
```

Appendix 2 Stopwatch for STFT

```
59      float[][] stft(Complex[] x, int wSamp)
60      {
61          int ii = 0;
62          int jj = 0;
63          int kk = 0;
64          int ll = 0;
65          int N = x.Length;
66          float fftMax = 0;
67
68          float[][] Y = new float[wSamp / 2][];
69
70
71
72          for (ll = 0; ll < wSamp / 2; ll++)
73          {
74              Y[ll] = new float[2 * (int)Math.Floor((double)N / (double)wSamp)];
75          }
76
77          Complex[] temp = new Complex[wSamp];
78          Complex[] tempFFT = new Complex[wSamp];
79
80          for (ii = 0; ii < 2 * Math.Floor((double)N / (double)wSamp) - 1; ii++)
81          {
82
83              for (jj = 0; jj < wSamp; jj++)
84              {
85                  temp[jj] = x[ii * (wSamp / 2) + jj];
86              }
87
88              tempFFT = fft(temp);
89
90              for (kk = 0; kk < wSamp / 2; kk++)
91              {
92                  Y[kk][ii] = (float)Complex.Abs(tempFFT[kk]);
93
94                  if (Y[kk][ii] > fftMax)
95                  {
96                      fftMax = Y[kk][ii];
97                  }
98              }
99
100
101         }
102
103         for (ii = 0; ii < 2 * Math.Floor((double)N / (double)wSamp) - 1; ii++)
104         {
105             for (kk = 0; kk < wSamp / 2; kk++)
106             {
107                 Y[kk][ii] /= fftMax;
108             }
109         }
110
111         return Y;
```

Appendix 3 STFT Sequential Function

```
3 references
Complex[] fft(Complex[] x)
{
    int ii = 0;
    int kk = 0;
    int N = x.Length;

    Complex[] Y = new Complex[N];

    // NEED TO MEMSET TO ZERO?

    if (N == 1)
    {
        Y[0] = x[0];
    }
    else{

        Complex[] E = new Complex[N/2];
        Complex[] O = new Complex[N/2];
        Complex[] even = new Complex[N/2];
        Complex[] odd = new Complex[N/2];

        for (ii = 0; ii < N; ii++)
        {

            if (ii % 2 == 0)
            {
                even[ii / 2] = x[ii];
            }
            if (ii % 2 == 1)
            {
                odd[(ii - 1) / 2] = x[ii];
            }
        }

        E = fft(even);
        O = fft(odd);

        for (kk = 0; kk < N; kk++)
        {
            Y[kk] = E[(kk % (N / 2))] + O[(kk % (N / 2))] * twiddles[kk * wSamp / N];
        }
    }

   return Y;
}
```

Appendix 4 freqDomain FFT

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

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

    Y = new Complex[nearest];

    Y = fft(compX, nearest);

    absY = new double[nearest];

    double maximum = 0;
    int maxInd = 0;

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

    for (int div = 6; div > 1; div--)
    {

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

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

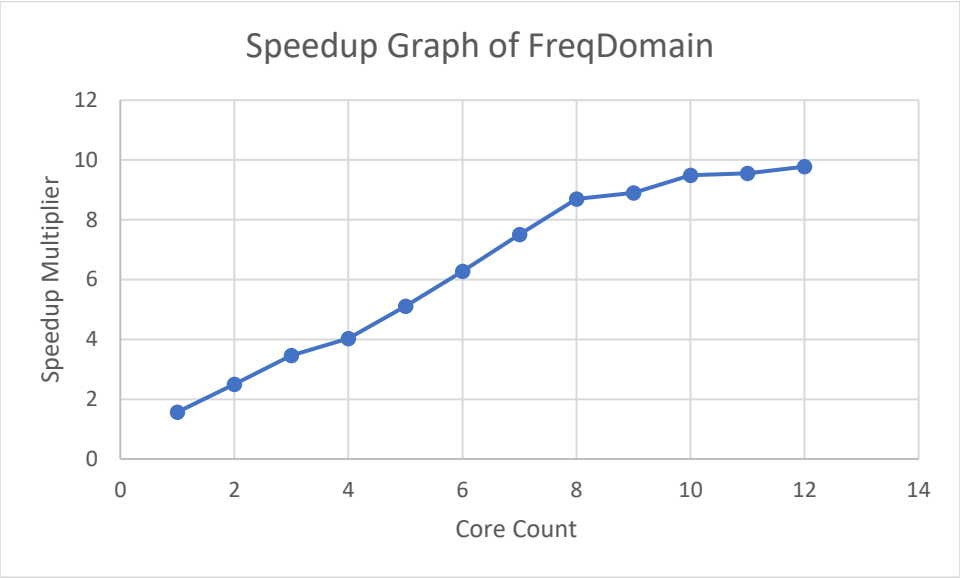
Appendix 5 onsetDetection For-loop

```
756         private Complex[] fft(Complex[] x, int L)
757         {
758             int ii = 0;
759             int kk = 0;
760             int N = x.Length;
761 
762             Complex[] Y = new Complex[N];
763 
764             if (N == 1)
765             {
766                 Y[0] = x[0];
767             }
768             else
769             {
770 
771                 Complex[] E = new Complex[N / 2];
772                 Complex[] O = new Complex[N / 2];
773                 Complex[] even = new Complex[N / 2];
774                 Complex[] odd = new Complex[N / 2];
775 
776                 for (ii = 0; ii < N; ii++)
777                 {
778 
779                     if (ii % 2 == 0)
780                     {
781                         even[ii / 2] = x[ii];
782                     }
783                     if (ii % 2 == 1)
784                     {
785                         odd[(ii - 1) / 2] = x[ii];
786                     }
787                 }
788 
789                 E = fft(even, L);
790                 O = fft(odd, L);
791 
792                 for (kk = 0; kk < N; kk++)
793                 {
794                     Y[kk] = E[(kk % (N / 2))] + O[(kk % (N / 2))] * twiddles[kk * (L / N)];
795                 }
796             }
797 
798             return Y;
799         }
```

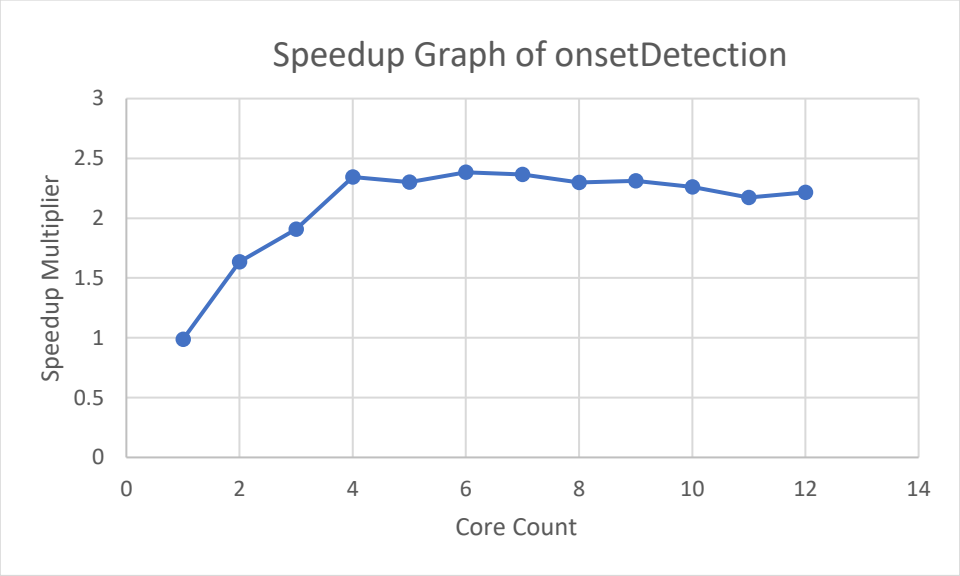
Appendix 6 onsetDetection FFT Method

Core Count	Total Time (%) Improvement	FreqDomain (%) Improvement	Onset (%) Improvement
1	1.229029177	1.568682754	0.987364621
2	1.994433876	2.504085376	1.635684027
3	2.487215535	3.467328562	1.907585004
4	2.982509746	4.031409396	2.345123258
5	3.214512832	5.114782016	2.301946344
6	3.508117487	6.275386544	2.383442266
7	3.670989496	7.5085	2.366258111
8	3.723135604	8.700463499	2.298721765
9	3.722156476	8.896327014	2.310860764
10	3.743816955	9.492414665	2.260719821
11	3.637964272	9.552798982	2.173869846
12	3.710446979	9.776692708	2.216444369

Appendix 7 Speedup Table of Parallelised Program



Appendix 8 Speedup Graph of FreqDomain



Appendix 9 Speedup Graph of onsetDetection

```python
import difflib
import os

main_path = os.path.dirname(__file__)
parrallel = os.path.join(main_path, 'DigitalMusicAnalysisParallel\\output.txt')
normal = os.path.join(main_path, 'DigitalMusicAnalysis\\output.txt')

print(main_path)

with open(parrallel) as file_1:
    file_1_text = file_1.readlines()

with open(normal) as file_2:
    file_2_text = file_2.readlines()

# Find and print the diff:
for line in difflib.unified_diff(
        file_1_text, file_2_text, fromfile='file1.txt',
        tofile='file2.txt', lineterm=''):
    print(line)
```

Appendix 10 Diff Checker Python

```
1 reference
Complex[] fft(Complex[] temp, Complex[] tempFFT)  // uses FFTW
{

    using (var pinIn = new PinnedArray<Complex>(temp))
    using (var pinOut = new PinnedArray<Complex>(tempFFT))
    {
        DFT.FFT(pinIn, pinOut); // computes the fft using the FFTW algorithm instead of the recursive algorithm present, DFT.FFT is a C# wrapper that calls DLL's that contain the FFTW code
    }

    return tempFFT;
```

Appendix 11 FFTW Implementation

Bibliography

Kehtarnavaz, N. (2008, January 1). *Frequency Domain Processing*. Retrieved from ScienceDirect: https://www.sciencedirect.com/topics/engineering/short-time-fourier-transform