CAB401 Assessment 2 Report

Brent Morgan
n10215662

Table of Contents

Original Sequential Application

The chosen application to parallelize for this project is a Digital Music Analyzer running on C#. This application is made to help beginner violin players by giving them feedback when they don't have a teacher present. It does this by the user inputting an audio file of them playing a selected song as well as a xml file of the sheet music to compare it to. It then analyses the notes being played and graphically shows the user how far off each note they are, so that they can then improve and fix these errors.

This program operates using an application class, window class and 4 object classes. The class diagram of these can be seen in figure 1 below.

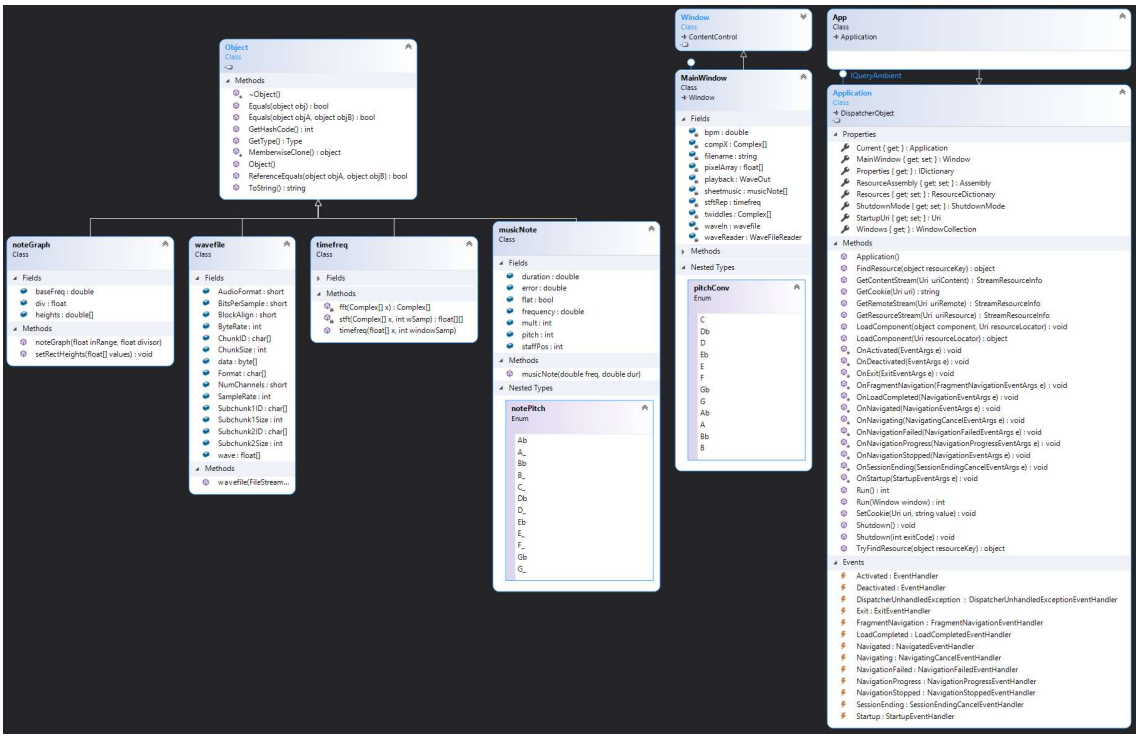

Figure 1 - Class Diagram

As seen in the figure, each class is decently complex with the number of fields, methods and properties present, because of this each of the classes will be explained individually.

Application Class

The Application Class is in the form of an App.xaml, this is the declarative starting point of the application and is automatically generated by Visual Studio alongside the Code-behind file called App.xaml.cs. The two of these files are partial classes which together work to allow for the coder to work in both markup and Code-behind. App.xaml.cs is an extension of the Application class, this meaning that .NET will refer to this class for startup instructions and then continue to launch the desired Window or page from there. Another important feature to occur here is the subscription to application events such as unhandled exceptions and application start (Tutorial, 2021).

MainWindow Class

The MainWindow class is where the bulk of the application is in terms of functions and creating the user interface. This is where timers can be applied to test the overall speed of the application and it most likely will be where there is room for improvement. This is the class with the necessary

functions to initialize the application, gather the users' inputs (.wav and .xml files) and then perform the music analysis on it.

musicNote Class

The musicNote Class is an Object Class with the function of identifying a musicNote from the inputs of the frequency and duration of the note played. This is for when the user inputs the audio (.wav) file of them playing the song so that the program can identify all the notes they are playing so that it can then later compare it to the actual note from the .xml file.

timefreq Class

The timefreq Class is where a lot of the mathematic computation occurs as it holds the Short-time Fourier transform and the Complex Fast Fourier transform functions. The Short-time Fourier transform (STFT) function computes the Fourier transform of partitions of a signal, its main purpose is generally noise reduction, pitch detection and pitch shifting (Wolfram, 2021). Then the Fast Fourier transform (FFT) converts the audio signal into individual spectral components which can then be used to provide information about the signal's frequency. The signal is samples over a period and then divided into its set frequency components which are signal sinusoidal oscillations at distinct frequencies, all of which having their own phase and amplitude (PCMag, 2021).

wavefile Class

The wavefile Class is an object class used to read the users inputted audio file (.wav) for it to then be used in the MainWindow class for analysis.

noteGraph Class

Finally, the noteGraph Class is another object class, but this time it is used when making the actual note graph that the user sees after their audio is analyzed which shows how close their played notes were to the actual song's notes. This class is used to define the shape of the notes to be used in this user-friendly graph.

Potential Parallelism Analysis

To determine the potential of parallelizing the program, the overall performance as well as the performance of individual parts of the code needs to be investigated. In terms of the overall runtime, a stopwatch was created in the MainWindow.xaml.cs file encapsulating the loadWave, freqDomain, readXML, onsetDetection, loadImage, and loadHistogram functions . This is because this is the bulk of the computing parts of the code as before this section, it is just opening the .wav and .xml files, and after it sis just producing an output for the user to see as well as the playback of the user's audio. This meaning that if there were any parts of the code that could be paralyzed it would be in those functions. In terms of the run time of these functions all together, the stopwatch timed it to be 3698ms. However, with separate stopwatches placed around each of the functions, it was found that the bulk of this processing time (99% of the 3698ms) was in the freqDomain and onsetDetection functions, this being with the freqDomain execution time taking 1934ms and the onsetDetection execution time taking 1727ms. Therefore, both will be further investigated for a potential parallelization area.

freqDomain Function

To isolate where the bulk of the processing time is in this function, two individual stopwatches were created around certain parts of the code measuring each parts processing time.

As you can see in Appendix 1.0 which shows this portion of code, the first stopwatch titled "timefreqStopwatch" was put around the timefreq function call, and then the second stopwatch titled "forloopStopwatch" was put around the for loop in the function. In this specific run, the overall freqDomain functions execution time was 1900ms, and inside this the timefreq function call took 1876ms, meaning that the for loop only took 15ms. This meaning that since the timefreq function call takes almost 99% of the execution time of freqDomain, this function should be further investigated for potential parallelization.

The freqDomain is a call to the timefreq function in the timefreq object Class, in this function there is multiple Fourier transforms which is the probable cause of the large execution time. At the end of the timefreq function script there is a call to the stft function (Short-time Fourier transform), which is defined later in the class, and inside this stft there is a call to the fft function (Fast Fourier transform) also defined in the same class. Due to the amount of for loops used inside the stft function, the fft function is called 2327 times inside it, so even through the fft processing time is quite small, the overall total of time it is running in the stft was found to be 1723ms using stopwatches. Then when looking at the total stft execution time of 1819ms, we can see that the fft function calls take up around 95% of the computation time of stft. Therefore, this for loop that the fft function is inside (can be seen in appendix 1.1) as well as the actual fft function (appendix 1.2).

onsetDetection Function

It was found inside a big for loop within the onsetDetection function that a Fast Fourier transform (fft) function identical to the one inside the freqDomain object Class is called 72 times, this totaling to 1256ms of computing time of the total 1687ms that the onsetDetection function takes. Therefore, similarly to the conclusion derived from the freqDomain investigation, the fft function or possibly the call to it should be investigated in terms of potential parallelization as if this is possible. Most likely the for loop that the fft function is called in will be parallelized as this will significantly drop the processing time of the overall program. The for loop in question is the one shown in appendix 1.3.

Mapping computation and/or data to Processors

In the previous section it was determined that the two functions with the highest computation time was the stft function inside the timefreq class, and the onsetDetection inside the MainWindow. The first of the code sections to be parallelized will be the for loop inside the stft function in the timefreq class. The standard for loop will be replaced with a Parallel.For loop, when doing this several errors emerge due to multiple threads accessing the same variable at once and overwriting it. To solve this, each variable that was being written to needed to be re-initialized inside the for loop so that each threads had their own instance of the variable. Another error that was found was that the fftMax variable which was the main output of this for loop had several versions due to there being several threads, but the end of this code section needed to produce just a single fftMax. To solve this issue, instead of each thread writing to a single variable, they wrote to an array initialized outside the Parallel.For loop and at the end each of the threads fftMax was compared to one another to get the biggest one and that was used as the overall fftMax variable. The code for the new parallel for loop can be seen in appendix 1.4.

Next, the for loop that had the fft call inside MainWindow.xaml.cs was next parallelized. A similar process was taken as the last part of code in terms of replacing the for loop with a Parallel.For loop. The same altercation in terms of the threads overwriting the variables occurred, so the same fix of re-initializing all the variables that get values given to it inside the loop. On top of this there was an error with the order the pitches were being added into the pitches list since the threads were running in a random order. To fix this, a pitches array was initialized and used inside the for loop when putting the pitches in order as it used the thread count (the same loop count) to put the pitch in a certain spot in the loop. Then after that, the pitches from the array were added in ascending order to the original pitches list giving the desired order. The code snippet for this can be found in appendix 1.5.

Timing and Profiling Results

To get the timing results for this application, several stopwatches were utilized in targeted parts of the code to see how long the processing time of each were. The three main stopwatches focused on were the overall programs stopwatch, a stopwatch that timed the freqDomain function, and lastly a stopwatch that timed the onsetDetection function.

Time for original sequential application

Using these three stopwatches, the following timings were found as an average of 10 runs on the original sequential application.

Overall Execution Time: 3537ms

freqDomain Execution Time: 1795ms

onsetDetection Execution time: 1706ms

Time for optimized parallelized application

To find which is the best of the optimized parallel times in terms of the number of processors/threads used, an average for each of the number of processors from 1-12 was found each having 3 runs. This data was then used to calculate the speedup of each number of processor and made into the speedup graph shown below in figure 2 (The raw data can be seen in appendix 1.6).

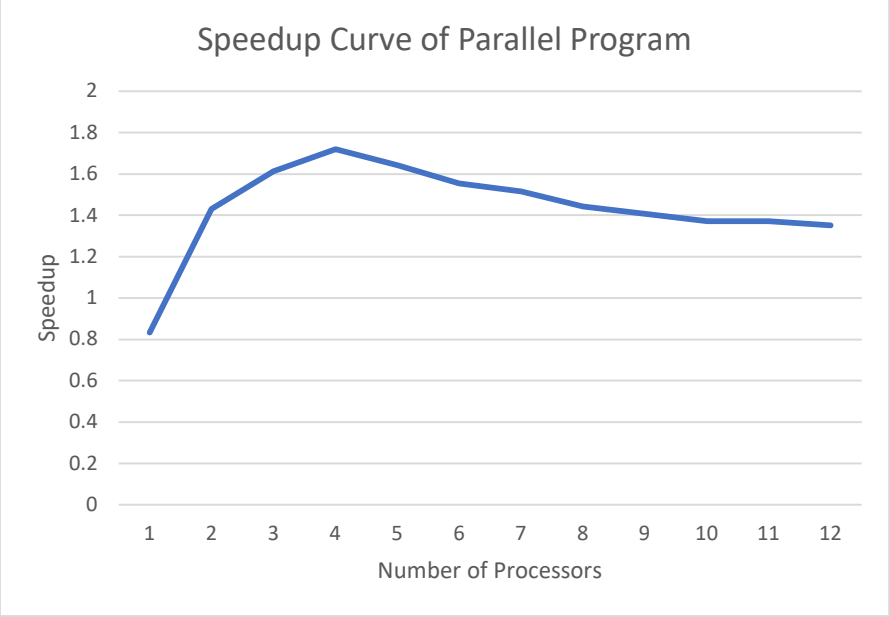

Figure 2 - Speedup Curve

What we can determine from this speedup curve is that from around 1-5 processors it follows the trend of a "typical" sub-linear speedup but then the trend falls after this. One thing we can derive from the graph is that the ideal number of processors for the best speedup is 4. Therefore, the average from the timers for this configuration of the parallelized code is shown below.

Overall Execution Time: 2057ms

freqDomain Execution Time: 1031ms

onsetDetection Execution time: 1026ms

Speedup Difference:

Now that we have the fastest parallel and sequential application times, we can calculate the difference in the three main timers.

Overall Execution Time: 1480ms

freqDomain Execution Time: 764ms

onsetDetection Execution time: 680ms

From the overall execution speedup, we can use the following formula inputting the overall times of the sequential and parallel versions.

Calculated Speedup: $\frac{execution\ time\ of\ best\ sequential\ program}{execution\ time\ of\ parallel\ program} = \frac{3537ms}{2057ms} = \mathbf{1.72}$

Profiling Results

The chosen profiling application for this project was JetBrains dotTrace as it works well with .NET programs. The main reason to use the profiler at this stage in the project where everything it already worked out is to just make sure that certain functions are being accessed by different threads, and we can see this to be true in figure 3 below showing that in the onsetDetection function that fft is being accessed multiple times all with multiple threads. This being the desired result of the profiling.

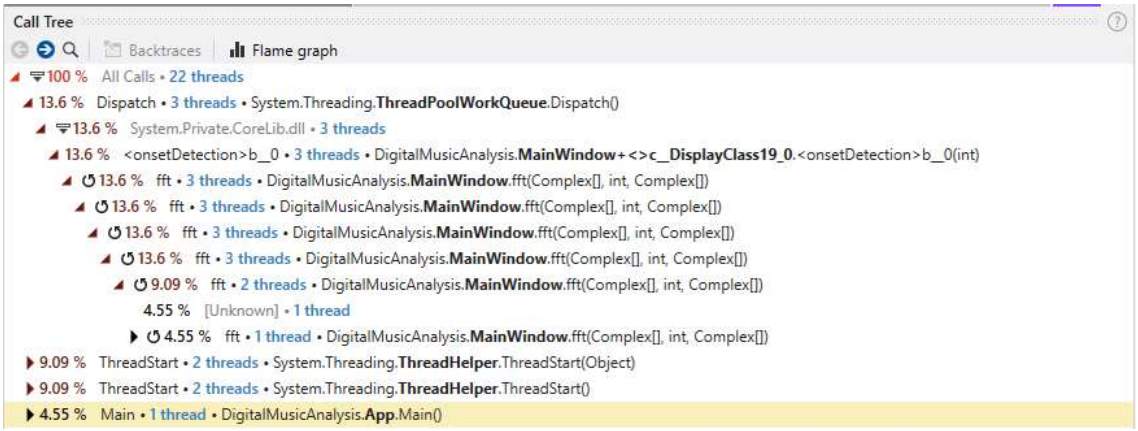

Figure 3 - Call Tree from JetBrains dotTrace

The final check is to make sure that the Process Memory usage is to that of a reasonable standard, and as you can see that for the most part it is a quick jagged increase then stabilizes out. The fact that it stabilizes out is the important part of this and is desirable. The correlating screenshot can be seen below in image 4.

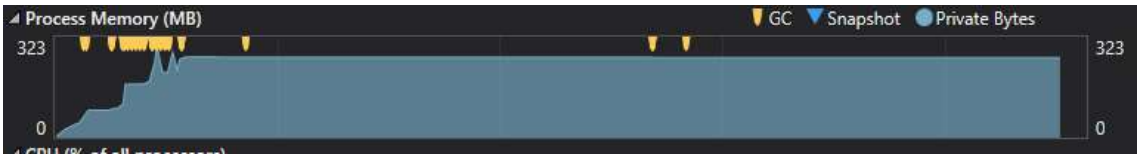

Figure 4 - Process Memory of running program

Parallel Version testing

To ensure that the parallel version of the application produced the exact same output as that of the sequential version, the values of the pitches data were recorded and compared. This was done by creating a string output function for musicNote in its class that returns a string containing all its variables. These variables being pitch, duration, flat, error, staffPos, mult, and frequency. This function was then called inside a for loop for all the music notes inside the alignedNoteArray and outputted to a text file using TextWriter. This process was done for both the sequential and parallelized projects to make sure all the notes were the same, which they were. To demonstrate this, the outputs of the text files were put into an online web source called diffchecker on https://diffchecker.com which analyses two texts and spots the differences.

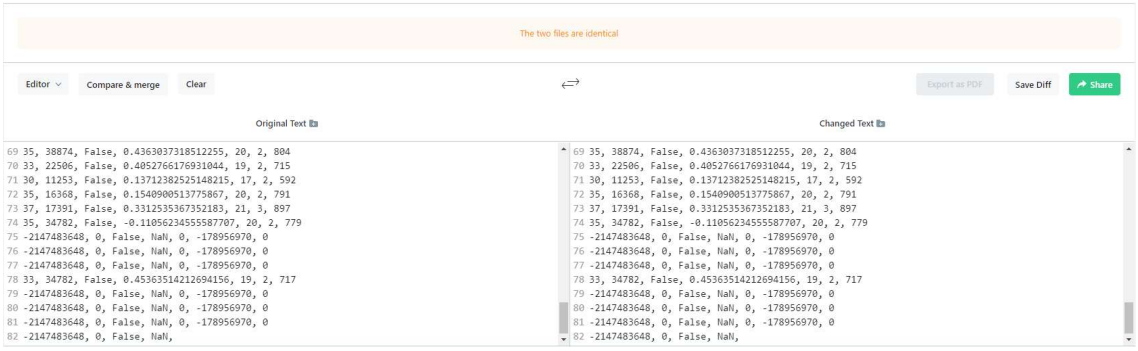

Figure 5 - Screenshot of diffchecker

The text file of the sequential is on the left and the parallel on the right. When running this application, it returned with the text at the top that stated “The two files are identical” meaning that the outputs for the two versions of the application are the exact same.

Another way to check would be to look at the graphical output from the function in the frequency and staff data visualizer. Below in figure 6 is the frequency data visualizer of the sequential code (left) and the parallel code (right). As you can see there is no difference in the two images meaning the parallelization was done correctly.

Sequential: **Parallel:**

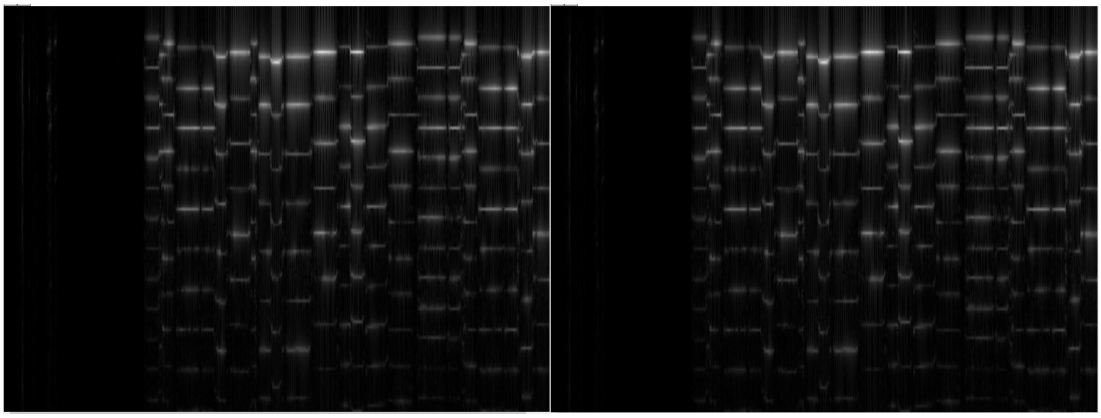

Figure 6 – Sequential & Parallel frequency data visualizers

Finally, to make sure, the start and the end of each of the staff data visualizers were compared, the first set belongs to the sequential code and the second set belongs to the parallelized version (figures 7 and 8). As you can see like in the frequency data visualizer, they are identical.

Sequential:

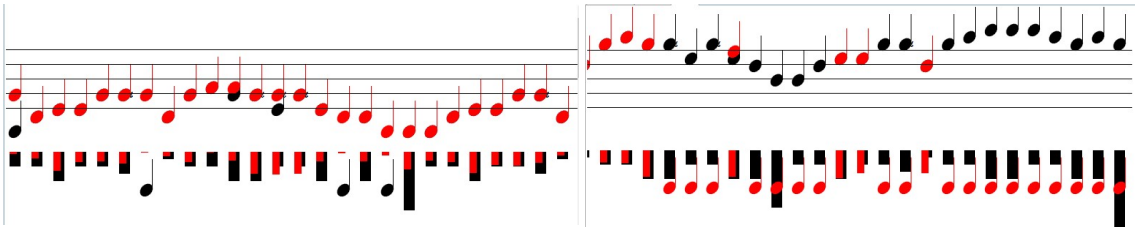

Figure 7 - Sequential staff data visualizer

Parallel:

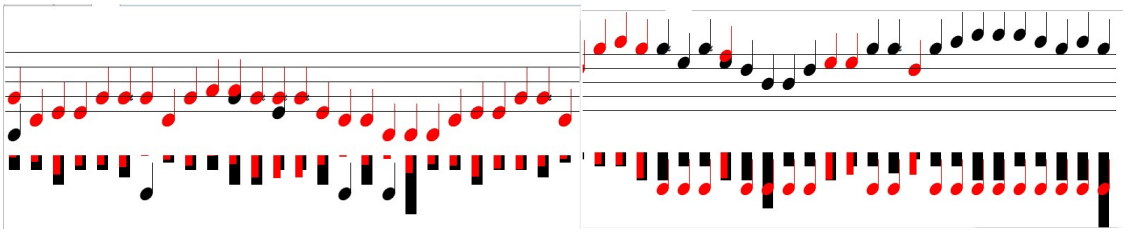

Figure 8 - Parallel staff data visualizer

After these 3 checks, it is safe to say that the parallel version of the code is sucessful in producing the exact same result in less time. Therefore making this overall parallelisation sucessful.

Description of compilers, software, tools, and techniques used

The IDE for this project was Visual Studio 2019, this is because it was a C# application and Visual Studio is the most supported and functioning IDE for C# applications. Using different libraries in Visual Studio the task of parallelizing the code was able to be done as well as the use of different applications so testing could be done. The ones that needed to be added from the original sequential program are as follows.

System.Diagnostics

The System.Diagnostics library was imported for the purpose of being able to add stopwatches in certain parts of the code and measure their runtimes. This was to be able to see how long certain functions such as onsetDetection or the whole application takes to run. Most importantly these times were recorded and then compared to the ones acquired for the same sections of code in the parallel version.

System.XML

The System.XML library was used for the purpose of being able to use a function called StreamWriter. This was described in the last section for the purpose of recording the data of the pitches in a text file for both the sequential and parallel version outputs for comparison.

System.Threading

Lastly, and most importantly was the System.Threading import which allowed for parallel functions to be called. The one used from this library was the Parallel.For which is a replacement to the standard C# for loop but instead it runs on multiple threads instead of just one like the normal for loop does.

JetBrains dot Trace

JetBrains dot trace was used for the profiling of the report as it is good for .NET programs in terms of being able to capture running code and show which parts utilize however many threads. This making it ideal for a program running parallel on multiple threads.

Overcoming performance problems and barriers

There were several barriers that needed to be overcome during the parallelization of this application. The biggest of which was data dependencies when trying to parallelize for loops. This was because when parallel for loops run, there is no way to organize what threads run in which order and therefore if you're adding a value into a list the order the variables get added are all in random orders. The way this was resolved in this case was in the onsetDetection function, instead of each run of that for loop adding the pitches value directly into the pitches list, a pitches array was initialized just before the for loop for these variables to be added to; this is because in an array you can easily say what position for the value to get added to, so in this case it added the value in the spot of the iteration number of the parallel for loop, this meaning they were all in the correct order. After this another for loop needed to be made outside separate to the parallel one to add each of the pitches in the newly created array into the original pitches list for it to be used later.

Another issue that arises when doing a parallel for loop is that the different threads are constantly overwriting variables if they are declared outside of the for loop. This is because they are all running at the same time and accessing the same instance of a variable. To fix this, each variable that was being written to (apart from arrays/lists to be used outside the for loop) was re-initialized inside the for loop, fixing the issue of all the threads overwriting each other's values.

Explanation of code

The first part of the code to be explained is the parallelization of the for loop inside the stft function. The sequential and parallel versions are shown side by side below with the changes done inside the parallel code outlined in red and then explained in order from top to bottom.

Sequential Code:

```
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];
        }
    }


}
```

Parallel Code:

```
float[] fftMaxArray = new float[(int)(2 * Math.Floor((double)N / (double)wSamp)) - 1];

Parallel.For(0, (int)(2 * Math.Floor((double)N / (double)wSamp)) - 1, ii => {

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

    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] > fftMaxArray[ii])
        {
            fftMaxArray[ii] = Y[kk][ii];
        }
    }
});

// Find the largest of the fftMaxArray and assign to fftMax
for (int iii = 0; iii < ((int)(2 * Math.Floor((double)N / (double)wSamp)) - 1); iii++) {
    if (fftMaxArray[iii] > fftMax) {
        fftMax = fftMaxArray[iii];
    }
}
```

Figure 9 - Sequential and Parallel Code Snippets showing changes

Changes:

1. An array called fftMaxArray is initialized outside the for loop. This is because each thread will have its own instance of the final fftMax variable, but at the end of this process there can only be one which is the largest of them. Therefore, an array was created to store all the threads fftMax value to then compare later.
2. The standard for loop is replaced with a Parallel.For loop for the purpose of assigning iterations to different threads.
3. In the sequential code the temp and tempFFT Complex arrays were initialized outside the loop because the array was able to be accessed one at a time. However, in the case of the parallel version it needed to be re-initialized inside the loop due to the threads constantly accessing this variable, so they need their own instance or else different threads would write values it in the overall one which didn't corelate to that instance.
4. Next, in the inner for loop call, the integers of jj and kk needed to re-initialized so that each thread has an instance that couldn't be overwritten by other threads (2 changes).
5. The next change in the code is simply replacing the fftMax variable from the sequential with the fftMaxArray that was made at the start of the code, this is to ensure that the variable being compared in the if statement is the fftMax value of the current thread running.
6. The next change is instead of defining what the value of fftMax is like in the sequential version, the fftMax of that thread is added into the appropriate spot (run number) fftMaxArray to be compared to the rest later.
7. The last change in this part of code is the extra for loop outside the parallelized for loop. The purpose of this section is to find which of the values in the fftMaxArray is the largest and then assign that value to the fftMax value as this single value is what's needed for the next section of code.

The next section of code to be explained is the for loop inside onsetDetection that was parallelized. The changes are highlighted in red then explained below in order of top to bottom.

Sequential Code:

```
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);
    }
}
```

Parallel Code:

```
Parallel.For(0, lengths.Count, mm =>
{
    int nearest = (int)Math.Pow(2, Math.Ceiling(Math.Log(lengths[mm], 2)));
    Complex[] twiddles = new Complex[nearest];
    for (int ll = 0; ll < nearest; ll++)
    {
        double a = 2 * pi * ll / (double)nearest;
        twiddles[ll] = Complex.Pow(Complex.Exp(-i), (float)a);
    }
    Complex[] compX = new Complex[nearest];
    for (int kk = 0; kk < nearest; kk++)
    {
        if (kk < lengths[mm] && (noteStarts[mm] + kk) < waveIn.wave.Length)
        {
            compX[kk] = waveIn.wave[noteStarts[mm] + kk];
        }
        else
        {
            compX[kk] = Complex.Zero;
        }
    }
    Complex[] Y = new Complex[nearest];
    Y = fft(compX, nearest, twiddles);
    double[] 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)
    {
        pitchesArr[mm] = (nearest - maxInd) * waveIn.SampleRate / nearest;
    }
    else
    {
        pitchesArr[mm] = maxInd * waveIn.SampleRate / nearest;
    }
});

//Add pitches arrays together
for (int j = 0; j < lengths.Count; j++) {
    pitches.Add(pitchesArr[j]);
}
```

Figure 10 - Sequential and Parallel code Snippets showing changes

Changes:

1. The standard for loop is replaced with the Parallel.For loop to assign iterations to different threads and decrease computation time.
2. Next, to avoid errors with data dependencies a separate instance of twiddles needed to be created for each iteration.
3. The same applies for the Complex arrays of compX and Y (next 2 changes)
4. After this, when calling fft there is an extra variable passed into it, this being the Complex twiddles array. This is because the fft function refers to this array but since there is a new array per thread, it needed to be parsed in so that the fft function was referring to the correct instance of the twiddles array.
5. For the same reason as the twiddles array change in number 2, the absY double array needed to be re-initialized inside the Parallel.For loop.
6. The next 2 changes are about added the pitches into an array instead of a list. This pitchesArr element was created before the for loop because if you were to use a list and the standard list.Add() function, the pitches would be added in a random order because all the threads are operating at random time intervals. Whereas when you do it in an array and place them in a set spot (Parallel.For loop iteration number), they will be in the right order.
7. The last change for this section is outside the Parallel.For loop and is just a small for loop to add each element from the pitches array into the pitches list to be used later.

Reflection

Throughout this project, I was able to better grasp concepts that I had learnt earlier in the semester. Applying these theories to my code and going through all the troubleshooting and debugging gave me a better understanding of parallel coding and just how it works.

While this parallelization was successful in terms of speeding up the program while still providing the exact same result and output to the user, it is likely that it could have been done better. More specifically, a greater speedup could have been obtained as the one obtained in this project was just 1.73 which while a solid improvement, isn't theoretically as optimized as it could be.

For this reason, I believe that I have not parallelized the code as well as possible. While I have parallelized the parts of the code I believed needed to be parallelized for an efficient speedup, it is likely that I missed other parts of code that could have benefited from parallelization.

If I were to restart the project, I would have spent more time profiling the code at an earlier time for me to be able to see exactly what parts of the code script took a lot of computation time and then continue to evaluate that sections possibility for parallelization. The reason this wasn't done for this project is due to an error when trying to use Intel VTune Profiler that didn't allow for this project to be profiled with this application. A lot of time was spent trying to get this to work due to it being one of the few compilers that showed the actual code snippets that take a lot of computation time when profiling; in comparison to the ones, I could get to work such as Visual Studio profiler and JetBrains dot Trace which did not show this.

This being said, a speedup of 1.73 is still good and most definitely noticeable to the user when the application is being run. For this reason, I believe the parallelization of this program was overall a success.

Appendix

Appendix 1.0 – freqDomain function:

```
1 reference
private void freqDomain()
{
    Stopwatch timefreqStopwatch = new Stopwatch();
    timefreqStopwatch.Start();
    stftRep = new timefreq(waveIn.wave, 2048);
    timefreqStopwatch.Stop();

    Stopwatch forloopStopwatch = new Stopwatch();
    forloopStopwatch.Start();
    pixelArray = new float[stftRep.timeFreqData[0].Length * stftRep.wSamp / 2];
    for (int jj = 0; jj < stftRep.wSamp / 2; jj++)
    {
        for (int ii = 0; ii < stftRep.timeFreqData[0].Length; ii++)
        {
            pixelArray[jj * stftRep.timeFreqData[0].Length + ii] = stftRep.timeFreqData[jj][ii];
        }
    }
    forloopStopwatch.Stop();

    Debug.WriteLine($"timefreq Execution Time: {timefreqStopwatch.ElapsedMilliseconds} ms");
    Debug.WriteLine($"for loop Execution Time: {forloopStopwatch.ElapsedMilliseconds} ms");
}
```

Appendix 1.1 – for loop inside stft function:

```
Stopwatch fftStopwatch = new Stopwatch();
int iterations = 0;

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];
    }

    fftStopwatch.Start();
    tempFFT = fft(temp);
    fftStopwatch.Stop();
    iterations += 1;

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

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

}
Debug.WriteLine($"fft Execution Time: {fftStopwatch.ElapsedMilliseconds} ms");
Debug.WriteLine($"Total times fft is called: {iterations}");
```

Appendix 1.2 – fft 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 1.3 – for loop inside onsetDetection function:

```
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];
    onsetDetectionfftStopwatch.Start();
    Y = fft(compX, nearest);
    onsetDetectionfftStopwatch.Stop();
    iterations += 1;
    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 1.4 – Parallel Version of for loop inside stft function:

```
float[] fftMaxArray = new float[(int)(2 * Math.Floor((double)N / (double)wSamp)) - 1];

Parallel.For(0, (int)(2 * Math.Floor((double)N / (double)wSamp)) - 1, ii => {

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

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

    tempFFT = fft(temp);
    float test = (float)Complex.Abs(tempFFT[500]);

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

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

// Find the largest of the fftMaxArray and assign to fftMax
for (int iii = 0; iii < ((int)(2 * Math.Floor((double)N / (double)wSamp)) - 1); iii++) {
    if (fftMaxArray[iii] > fftMax) {
        fftMax = fftMaxArray[iii];
    }
}
```

Appendix 1.5 – Parallel version of for loop inside onsetDetection function:

```
int iterations = 0;
double[] pitchesArr = new double[lengths.Count];

Parallel.For(0, lengths.Count, mm =>
{
    int nearest = (int)Math.Pow(2, Math.Ceiling(Math.Log(lengths[mm], 2)));
    Complex[] twiddles = new Complex[nearest];
    for (int ll = 0; ll < nearest; ll++)
    {
        double a = 2 * pi * ll / (double)nearest;
        twiddles[ll] = Complex.Pow(Complex.Exp(-i), (float)a);
    }
    Complex[] compX = new Complex[nearest];
    for (int kk = 0; kk < nearest; kk++)
    {
        if (kk < lengths[mm] && (noteStarts[mm] + kk) < waveIn.wave.Length)
        {
            compX[kk] = waveIn.wave[noteStarts[mm] + kk];
        }
        else
        {
            compX[kk] = Complex.Zero;
        }
    }
    Complex[] Y = new Complex[nearest];
    onsetDetectionfftStopwatch.Start();
    Y = fft(compX, nearest, twiddles);
    onsetDetectionfftStopwatch.Stop();
    iterations += 1;
    double[] 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)
    {
        pitchesArr[mm] = (nearest - maxInd) * waveIn.SampleRate / nearest;
    }
    else
    {
        pitchesArr[mm] = maxInd * waveIn.SampleRate / nearest;
    }
});

//Add pitches arrays together
for (int j = 0; j < lengths.Count; j++) {
    pitches.Add(pitchesArr[j]);
}
```

Appendix 1.6 – Data from getting the speedup on different numbers of processers

Number of Processors	Run #1	Run #2	Run #3	Average	Speedup
1	4022	3778	4926	4242	0.833805
2	2456	2547	2412	2471.667	1.431018
3	2199	2224	2159	2194	1.612124
4	2044	2089	2038	2057	1.719494
5	2129	2209	2125	2154.333	1.641807
6	2257	2344	2225	2275.333	1.554498
7	2356	2290	2356	2334	1.515424
8	2494	2455	2409	2452.667	1.442104
9	2465	2520	2556	2513.667	1.407108
10	2640	2474	2625	2579.667	1.371107
11	2595	2581	2560	2578.667	1.371639
12	2655	2555	2646	2618.667	1.350687

References

PCMag. (2021, 10 10). *FFT*. Retrieved from PCMag: https://www.pcmag.com/encyclopedia/term/fft

Tutorial, W. (2021, 10 09). *Working with App.xaml*. Retrieved from WPF Tutorial: https://wpf-tutorial.com/wpf-application/working-with-app-xaml/

Wolfram. (2021, 10 10). *Audio Short-Time Fourier Transform (STFT)*. Retrieved from Wolfram: https://www.wolfram.com/language/12/new-in-audio-processing/audio-short-time-fourier-transform-stft.html?product=language