1/11/2021

Julian Zhu

QUT Year3

CAB401 Parallelization Project

Digital Music Analysis

Contents

[Explanation of the original sequential application 2](#_Toc86635912)

[How does Digital Music work 2](#_Toc86635913)

[Potential parallelism 3](#_Toc86635914)

[Parallelism for timefreq.cs 4](#_Toc86635915)

[Parallelism for onsetdetection() 6](#_Toc86635916)

[Abstractions or programming language used in synchronization 7](#_Toc86635917)

[Result 7](#_Toc86635918)

[Timing 7](#_Toc86635919)

[Result comparison 7](#_Toc86635920)

[Compiler, software, tools and techniques. 12](#_Toc86635921)

[Compiler 12](#_Toc86635922)

[Parallelisation framework and Hardware 12](#_Toc86635923)

[Techniques 13](#_Toc86635924)

[Barriers 13](#_Toc86635925)

[Explanation Code 14](#_Toc86635926)

[timefreq.cs 14](#_Toc86635927)

[mainwindow.cs 17](#_Toc86635928)

[Reflection 18](#_Toc86635929)

# Explanation of the original sequential application

The application that is used for parallelisation for this project is “DigialMusic”. The application is downloaded on the QUT’s blackboard under assessment. The “DigialMusic” application is written in the C# programming language. The purpose of the application is to help beginner users identify which note is currently playing according to uploaded files. The application shows which note the student played and where the note should be playing at. The application also visualises the frequency in black and white on the first page of the GUI.

Once the user runs the application, the first action that the user must perform is to upload two types of files, “.wav” and “.xml”. The “.wav” is the file that the application will play through, and the “xml” file is the provided file (the file that the user played) for the application to compare. There are in total three tabs from the application, “frequency”, “octaves”, and “staff”. The “frequency” tab shows the audio frequency in black and white. The “octaves” tab shows which note is currently playing. Lastly, the “staff” tab shows whether if the student is playing correctly or not. The red notes mean that the note is incorrect, and the back notes mean the right note. The user can also see the percentage of pitcher error if the user hovers over the notes in the tab.

# How does Digital Music work

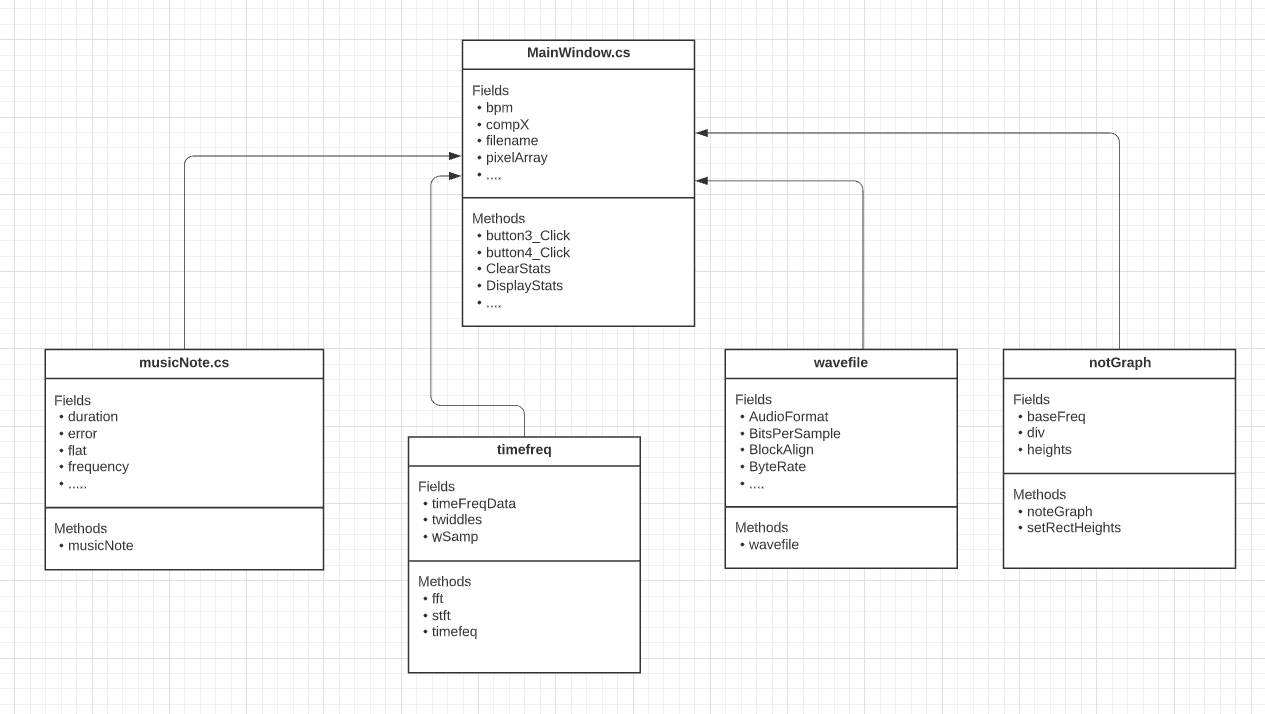


Figure 1 (class diagram)

The application contains multiple classes, those are musicNote.cs, noteGraph.cs, timefreq.cs and wavefile.cs. The musicNote class contains several logical switch cases, which helps to check the note's position; it also includes pitch of the note, duration, flat note, error, staff position, and frequency. The class noteGraph is used to set the height of the note. The third class, timefreq, is used to analysing frequency. Furthermore, The class contain complex algorithms, such as FFT(Fast Fourier transform) and STFT(Short-time Fourier transform). The last class wavefile is binary reader, the class can read and information from the file, including chunk size, format, audio format, number of channels, sample rate, byte rate, bites per sample, and many more. The main function is located in "mainWindow.xaml.cs" file. The file contains multiple functions in including loading file, processing file, and GUI design. The most important function will be analysed below.

1. “loadwave()” function

The first important function that the application will run once the user runs the application is “loadwave”. If the uploaded file is incorrect, the application will return an error message telling the user; else, the function will convert the uploaded file into an array.

1. “freqDomain()” function

The next function that will run after “loadwave()” function is “freqDomain()” function. This function will lead to a class called “timefeq”. The “timefeq” class has two functions called “stft” and “fft”, which stands for Short-time Fourier transform and Fast Fourier transform.

1. “readXML” function

The “readXML” function will read the uploaded xml file, and convert them in to lists.

1. “onsetDetection”

The “OnesetDecetion” function will check whether if the note starts playing and when did it stop.

The rest of the function such as “loadImage()”, “loadHistogram()”, and “PlayBack()” can be described by its function name.

# Potential parallelism

The method of parallelism will be used according to the Parallizaion Methodology table. The first four steps have already been done; the starting step will be to determine sections that could be parallelized by using a performance profiler.

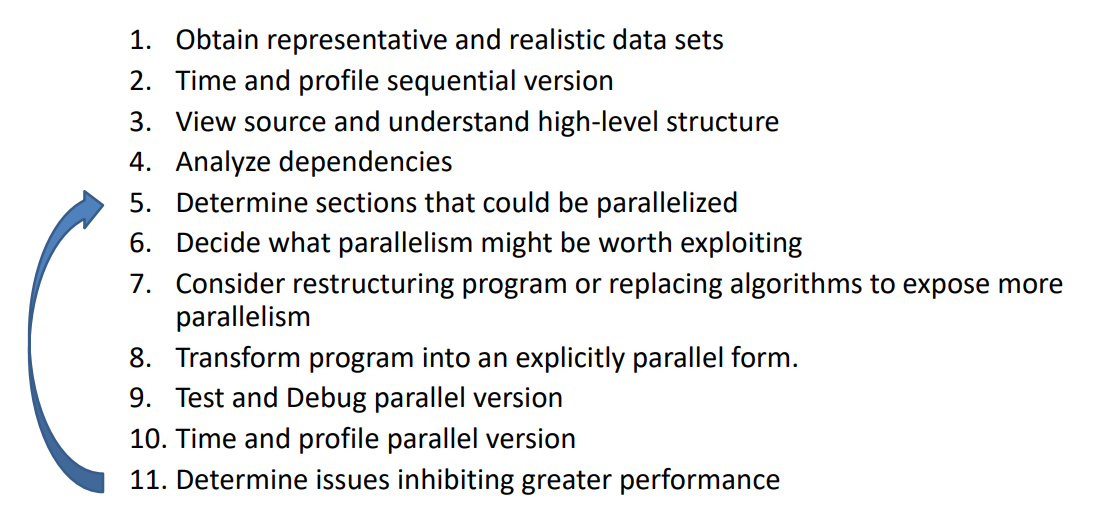


Figure 2(Parallizaion Methodology table from Wayne's Lecture)

The Performance profiler will capture the CPU Usage and give the user a great view of which part of the program is doing most of the work. Once I ran the performance profiler, it was clear that most calculation and time is on onsetDetection() and freqDomain(), with 28.06% and 27.68%. According to the profiler, it is believed that paralleling those two classes will significantly reduce the amount of time it takes, which result in speeding up the whole application.

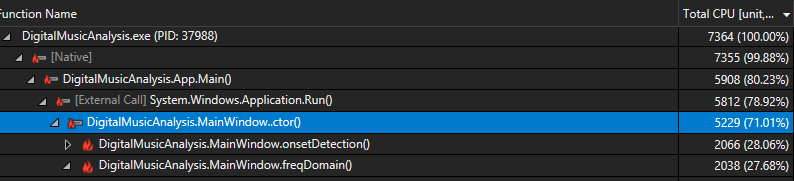


Figure 3 (Performance Profiler Capture)

## Parallelism for timefreq.cs

According to the performance profiler, 27.44% is at the class timefreq, and if we examine the class, we can see that out of 27.44%, there is 26.82% inside a function called stft.

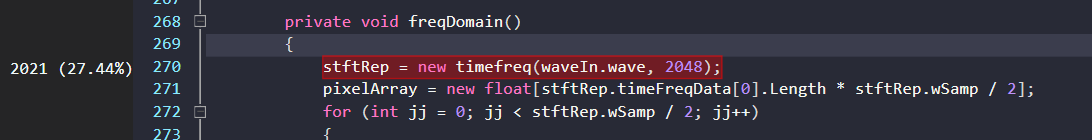


Figure 4(Workload from timefreq.cs)

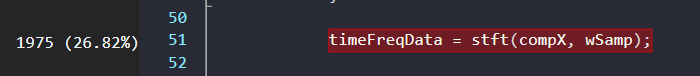


Figure 5(Workload from stft method)

By viewing inside of timefreq class, the first two loops take a bit of time, the first one is the for loop with the if-else statement, and the second one is just a for loop. Since there are no data dependencies for those two loops, it makes those loops safe to parallel. The method that I’ll parallel those loops is by using implicit threads because both of them are simple for loop with no data dependencies.

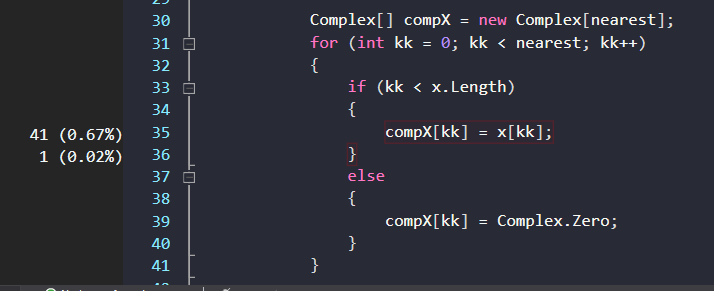


Figure 6(First loop from timefreq.cs)

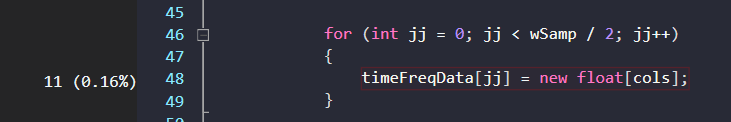


Figure 7(Second loop from timefreq.cs)

Implicit threads are high level libraries and language constructs built on top of more primitive threading libraries.

If we look down a little more, we can see where most of the work is done, which is a line calling the stft method.

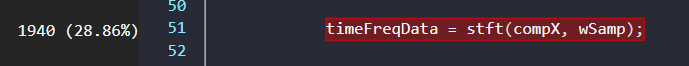


Figure 8 (Workload for stft method)

To parallel this line of code, we will require to check the method’s structure of stft. First, identify whether it is safe to parallel, then check if it requires changing the algorithm or transforming it to speed up.

Inside stft method, there are two for loops, one is a simple for loop, and one is a nested for loop.

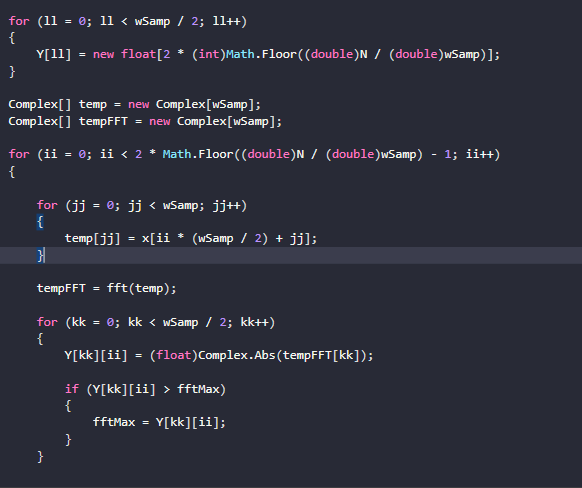


Figure 9 (loops inside stft method)

The first for loop is similar to the previous for loop; it is safe to parallel by simpling using implicit threads. However, the second loop is more complex. The second loop is a loop that calls the method FFT function where the calculation is being done. The function fft is a method that contains complex algorithms, and it is not safe to parallel due to data dependencies. The data output is dependent on the complex number array, which makes this function hard to parallel. The only way to parallel fft function is to transform the algorithms so that the data no longer depend on each other. I decided to parallel the loop that calls fft, due to the complexity of the transformation (more details can be read in reflect section). The parallel method that I decided to use for this loop will be Explicit threads. First, calculate blocksize, which is the end of the second loop(2\*Math.Fllow((double)N/ (double)wSam) -1), decided by the number of threads. Each thread will run a part of the loop, which archives in speeding up the application. I overcame the dependencies problem by putting the complex array into each thread instead of leaving it out of the loop

(More detail can be seen in the screenshot from the Explanation section)

## Parallelism for onsetdetection()

By viewing the profiler, it is clear to see the only loop that is worth to parallel is the largest for loop in onsetdecetion. There are couple varables that will require to set it to global, those are twiddles, compX, Complex[]Y, absY[], and int ll. The reason remains the same as the pervious why I put complex array in each thread, because of data dependency. Last change that will require to be make is to change the pitches list into array.

The only loop that I paralleled is the main for loop from the class, which does the most amount of work.

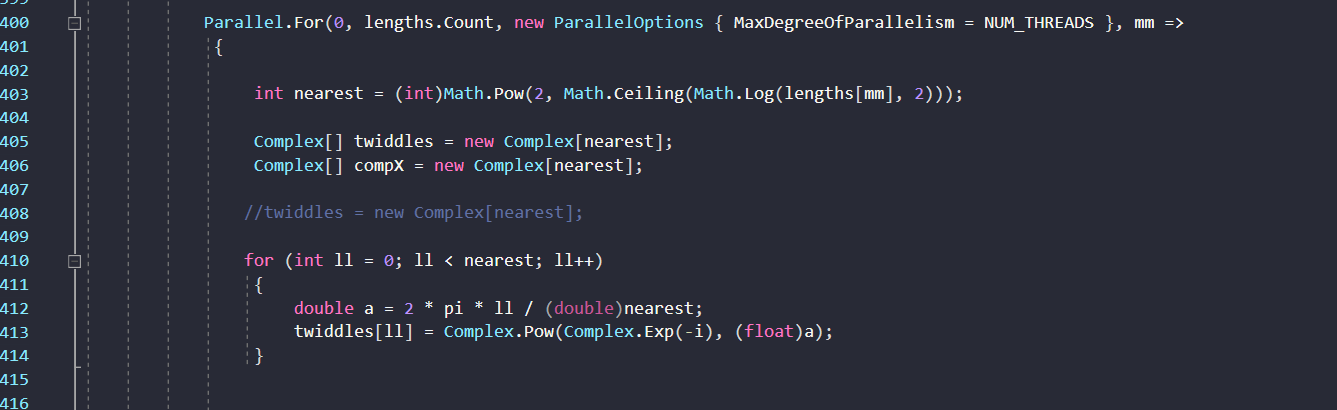


Figure 10(onsetdetection loops)

There are two FFT function can be find in application. One is located at the timefreq class, and one is at the mainwindow class. Combining two FFT function into one class will not result in application speed up. For the FFT funciton to speed up will require to change FFT’s alogrthim. Unforuntly, I did not manage to change the alogrhtim, therefore, I left FFT untouched for both classes. Lastly, there no sychronisation method that is used in both parallzaion, this is because that the parallel library from C# has pre-sychronisation builded into.

(Screen shots of the code after parallelsion will be shown on the Explanation secetion)

# Abstractions or programming language used in synchronization

The abstraction used throughout the application is called TPL, which stands for “Task Parallel Library”. The TPL parallelism abstractions contain several public types and APIs in the System.Threading, and System.Threading.Tasks. Once it is added, the user will be able to use commands that active threadings.

# Result

## Timing

The timing is done by using C# library called Diagnostic. Once the Diagnostic has been added, the function stopwatch will be set on three parts of the application. First timer will time the whole application, last two timer are at freqDomain and onsetdetection timer, which will time their own section. Lastly, two open file commands will not be times because it depend on the user’s interaction.

1. Whole application timer with number of processors

|  |  |  |
| --- | --- | --- |
| **Number of processors** | **Sequential Time (ms)** | **Paralleled Time (ms)** |
| 1 | 4357 | 5281 |
| 2 | 4622 | 3817 |
| 3 | 3907 | 2697 |
| 4 | 3911 | 2483 |
| 5 | 3794 | 2448 |
| 6 | 3817 | 2295 |
| 7 | 3837 | 2285 |
| 8 | 3829 | 2224 |
| 9 | 3812 | 2212 |
| 10 | 3751 | 2201 |
| 11 | 3784 | 2253 |
| 12 | 3739 | 2192 |

1. “freqdomain” timer with number of processors

|  |  |  |
| --- | --- | --- |
| **Number of processors** | **Sequential Time (ms)** | **Paralleled Time (ms)** |
| 1 | 2177 | 3011 |
| 2 | 2474 | 2000 |
| 3 | 2043 | 1390 |
| 4 | 2045 | 1239 |
| 5 | 1917 | 1180 |
| 6 | 1965 | 1135 |
| 7 | 1978 | 1190 |
| 8 | 1999 | 1008 |
| 9 | 1958 | 1063 |
| 10 | 1935 | 1044 |
| 11 | 1972 | 1073 |
| 12 | 1924 | 1091 |

1. “onsetdetection” timer with number of processors

|  |  |  |
| --- | --- | --- |
| **Number of processors** | **Sequential Time (ms)** | **Paralleled Time (ms)** |
| 1 | 2102 | 2147 |
| 2 | 2108 | 1770 |
| 3 | 1820 | 1271 |
| 4 | 1826 | 1207 |
| 5 | 1838 | 1231 |
| 6 | 1807 | 1124 |
| 7 | 1818 | 1057 |
| 8 | 1791 | 1179 |
| 9 | 1813 | 1112 |
| 10 | 1776 | 1118 |
| 11 | 1771 | 1143 |
| 12 | 1772 | 1062 |

Below is the speed up chart by using the formula, execution time of best sequential program, divided by execution time of parallel program.

## Result comparison

Both sequential application’s result and the parallel application’s result will be compared using the GUI interface. The GUI interface shows the audio frequency, current note playing, and a graph showing which note is playing incorrectly. Comparing the GUI result, it will give us whether the result is correct or not. If it is incorrect, the graph will look different from the sequential application’s graph (example in Barrier section). In addition, the application also provides string matching output, comparing output will also give the result on whether the application is correct or not.

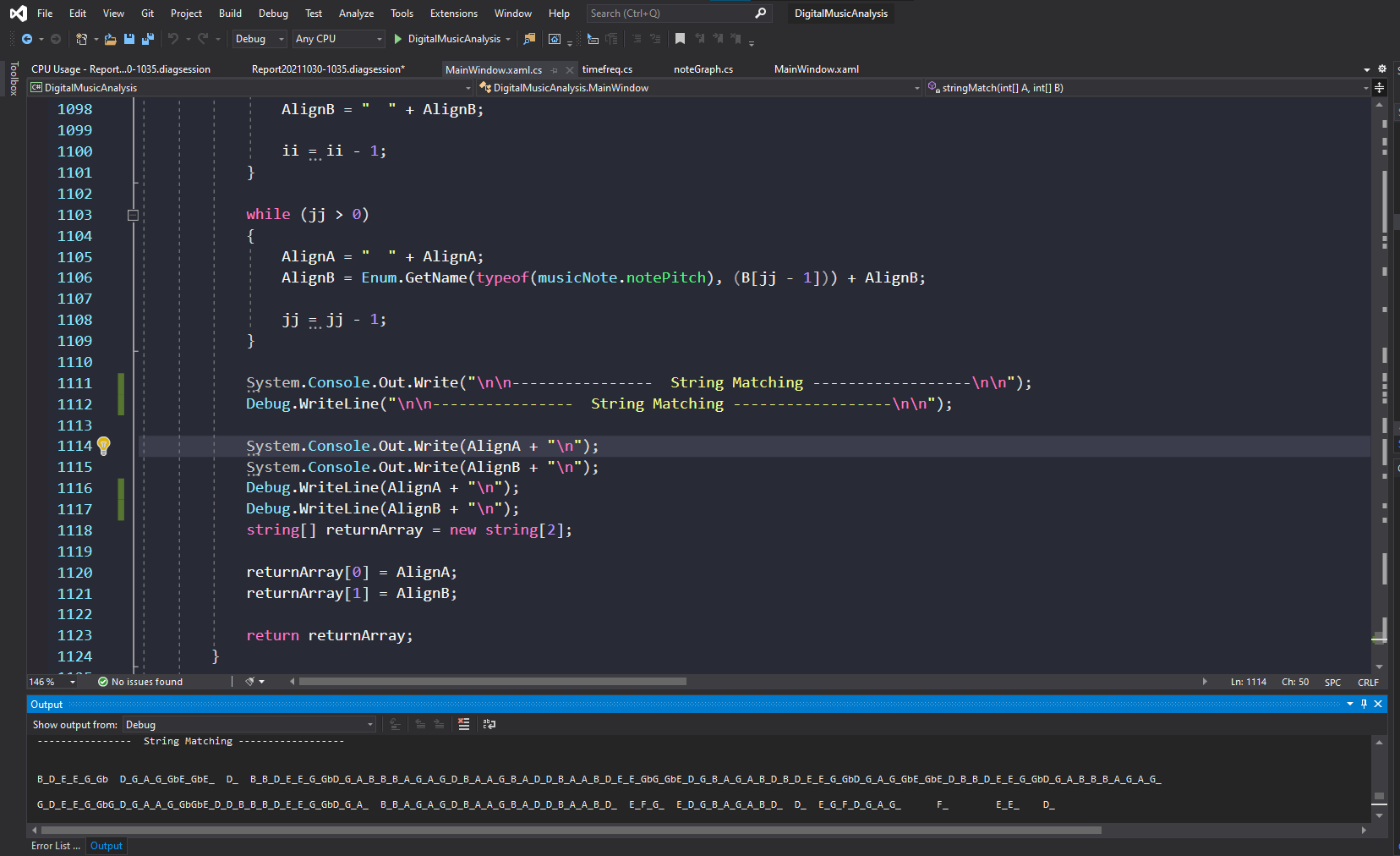


Figure 11(sequential output)

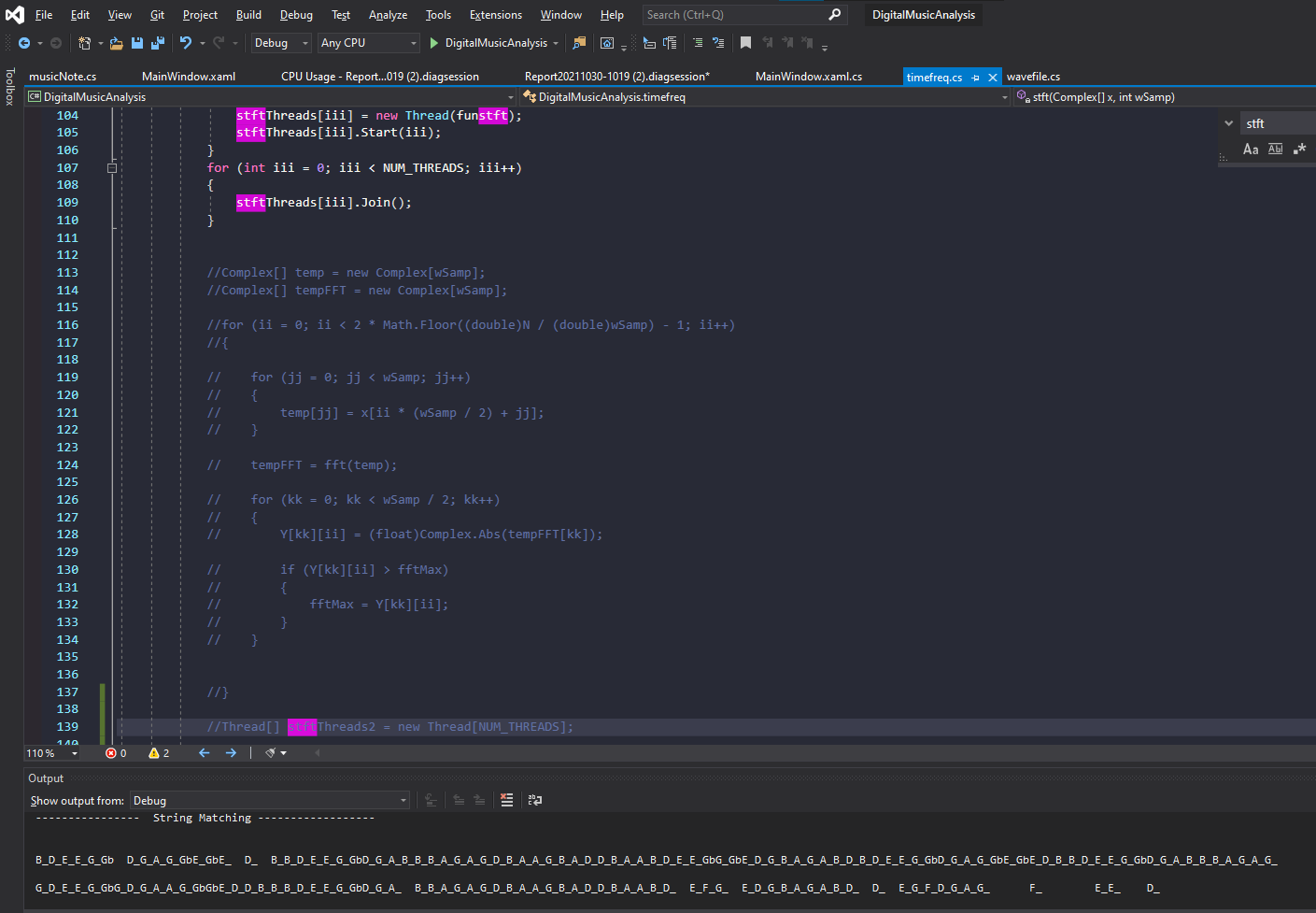


Figure 12(parallel output)



Figure 13(sequential frequency map)

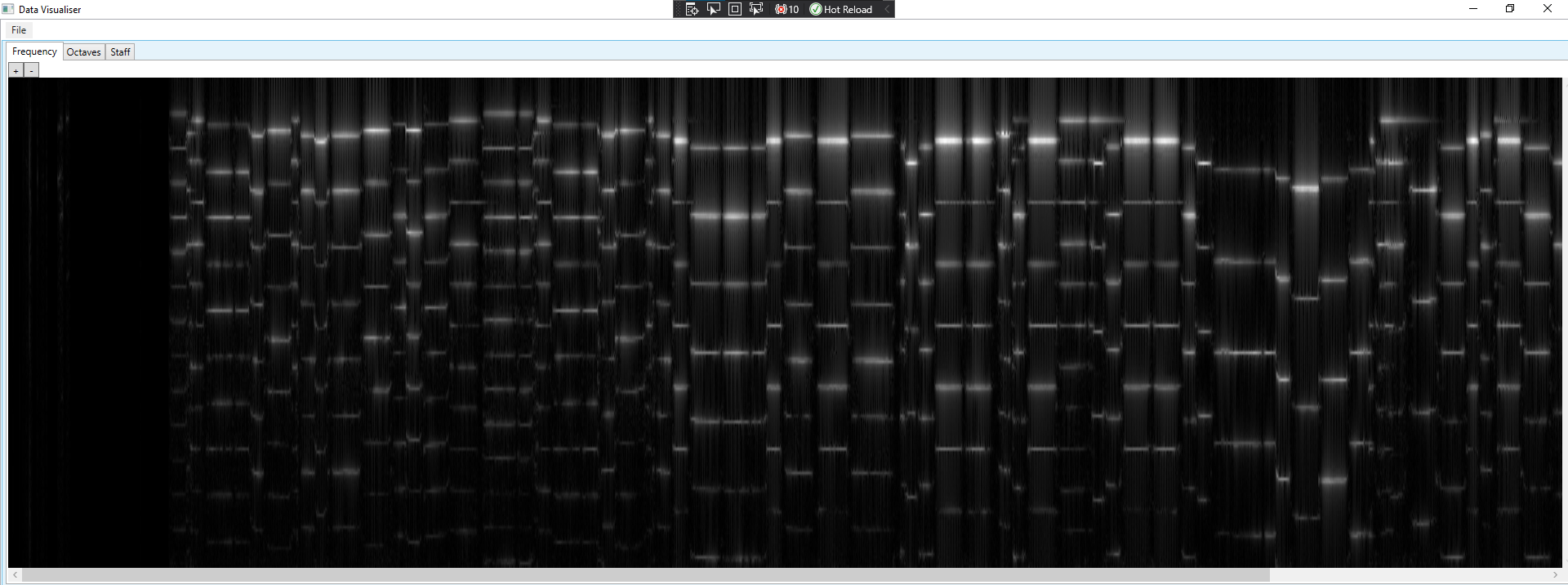


Figure 14(parallel frequency map)

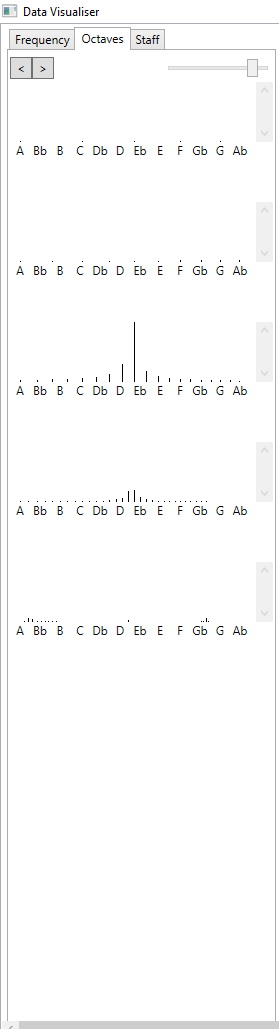
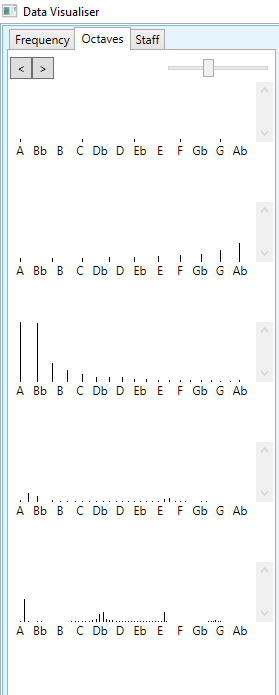
 

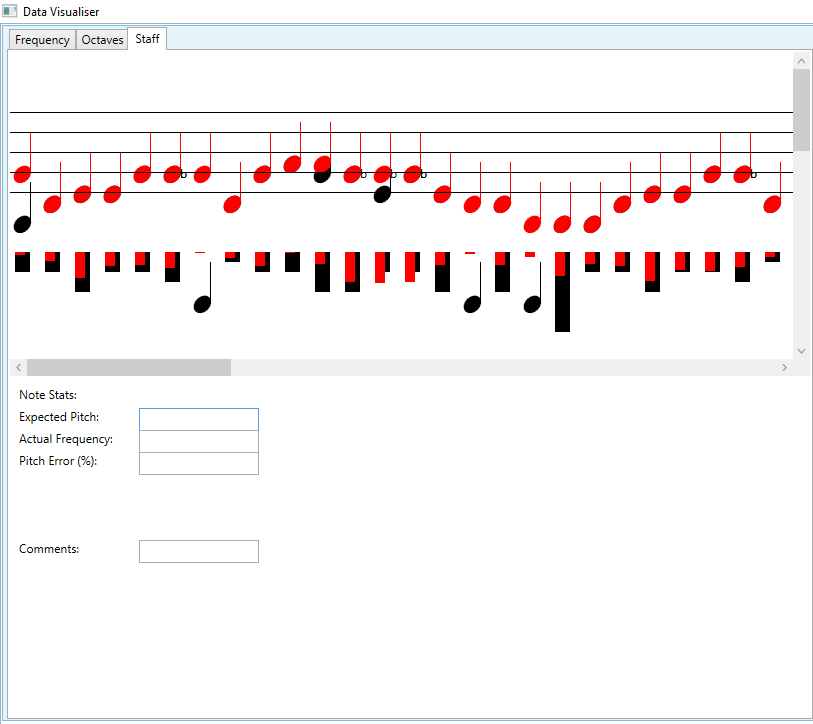
Figure 15(sequential note left, parallel note right)

Figure 16(sequential staff tab)

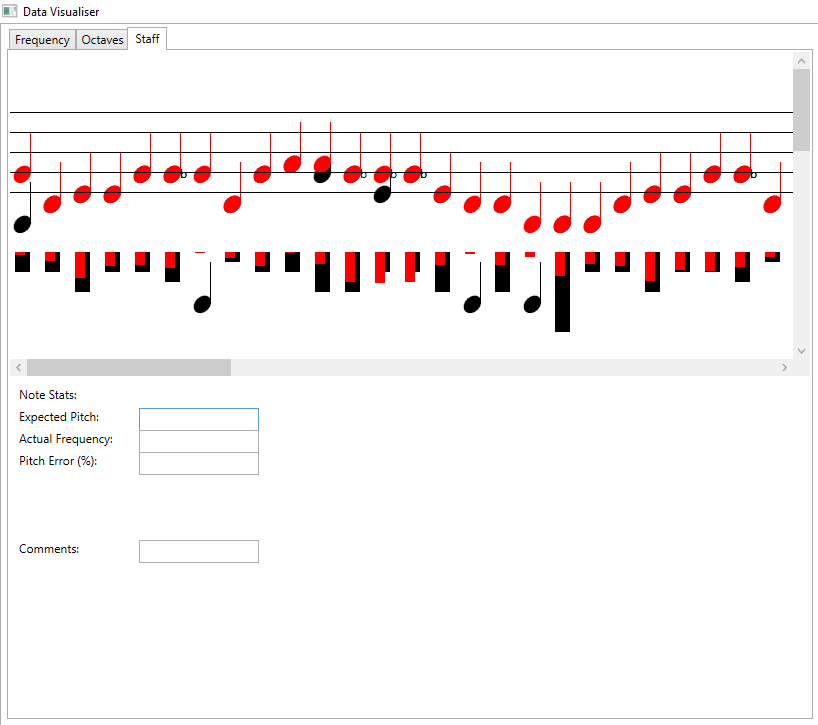


Figure 17(parallel staff tab)

According to the screenshots, both sequential application’s output and parallel application’s output look identical, making the parallel application’s result correct.

# Compiler, software, tools and techniques.

## Compiler

The compiler that is used for the application is the default compiler from C#. The user should change the number of processors he would like to use and then build the application before running in Debug mode; this is because my application does not produce any output when running in Release mode (discuss more in the barrier section).

## Parallelisation framework and Hardware

The application is running on my laptop. Below is the list of my laptop’s components

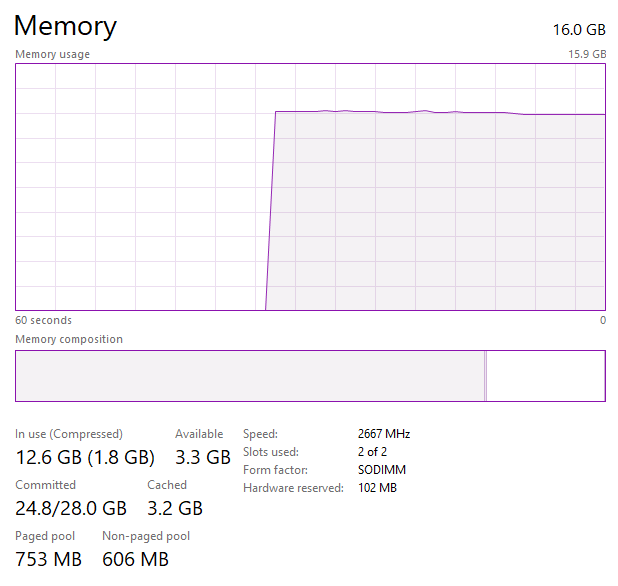
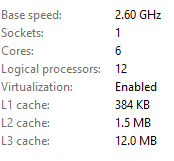


Figure 18(GPU, CPU, Ram, Core, and processor)

The application is built and written on C#, which makes C# is the frame work for parallelisation. The tool that was used for identify application preforming is a feature from Visual studio, called “Performance Profiler”. The performance profiler gives the user a visual graph on which part of the program is heavily used. The feature also has options to capture GPU usage and many others.

## Techniques

All the techniques and algorithms behind parallelism code are from Wayne’s lecture and the student presentation from Jamie. Both resources give me some great examples of how I should write my threads.

Jamie’s presentation gave some great examples of writing different types of threads. Wayne’s lecture helped me on how to approach application. Once I find the loops that have the potential to be parallel, I will follow the parallelisation logical steps. First is to determine what could be safely parallelised; this means I need to analyse control and data dependencies and decide whether the program needed a new algorithm or transform the algorithm.

# Barriers

The main barrier for this project is to understanding the application. The application contains many very complex algorithms and several loops. The barrier makes the starting of the project very time consuming and difficult. I managed to use high-level class diagram features from the visual studio to help me understand what kind of variables and method is included in each class. Once I understand what the majority of the variables do, the next step is to understand the algorithm. Unfortunately, the algorithm behind fft I did not fully understand; however, I found the data dependencies inside timefreq class and managed to parallel the loop that calls the fft function. This dramatically reduces the time that it takes to run the application.

The next barrier that I encountered is that the function console.writeline does not work on my application. Every time when I run the application, it shows number of error mages from the consoles, but not any console.writeline commands, this makes impossible for me to see any results. After I did some google research, it seems that the cost of this problem could be many, so I decided to avoid this problem and find an away around it, which is using Debug.writeline. This replacement makes the application only be able to print out lines when running in Debug mode, which makes the application slower then running in release mode.

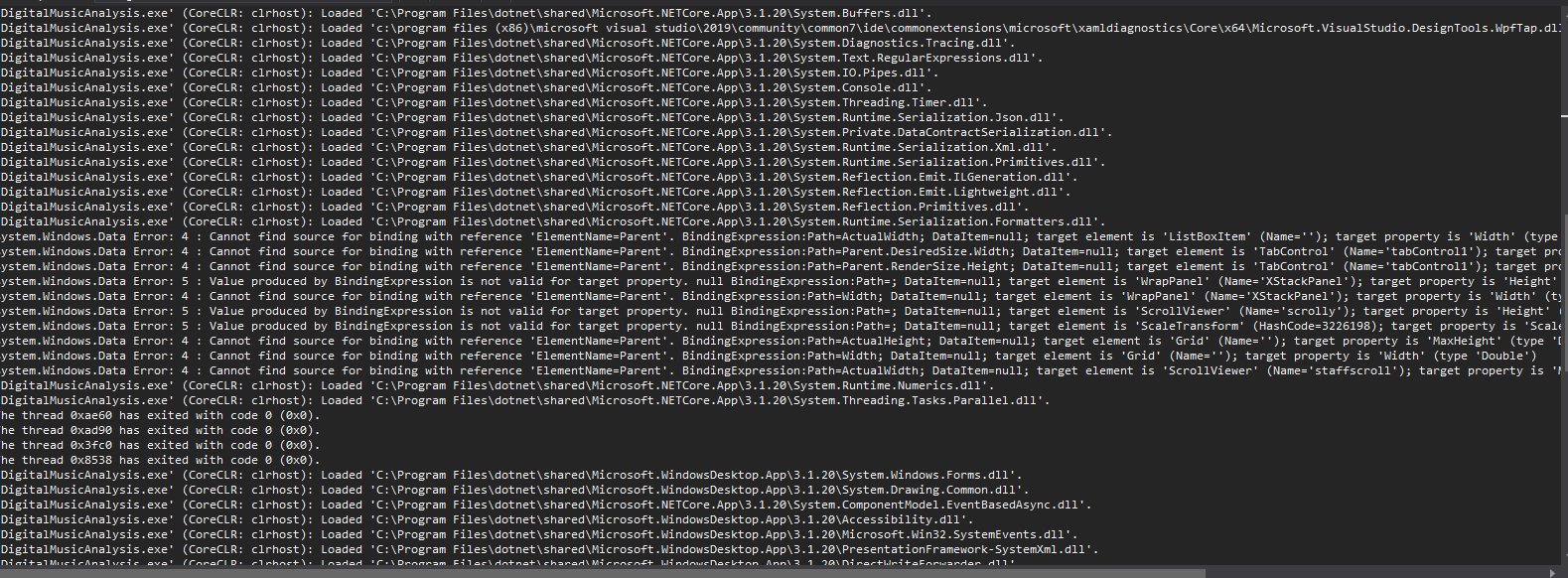


Figure 19(output error)

Furthermore, some paralleled code that was written resulted in slowing down the application. This happened to me while I worked on timefreq class. By revisiting the lectures, I found out why it is slowing down; it is because each "work unit' is not sufficiently large enough, which resulted in did not offset the overhead. The salutation of this problem is to find the loop that does the most amount of work by using a performance profiler and parallel the loops that are doing the most amount of calculation.

The last barrier I had while paralleling is that the output is entirely wrong compared to the sequential version. This is due to the part I tried to parallel having data dependencies. Two methods can potentially overcome this barrier: one is transforming the algorithm, and the second is creating a new algorithm that can reprice the current one. I manage to overcome this problem by transforming the algorithm by bringing the complex array into the thread

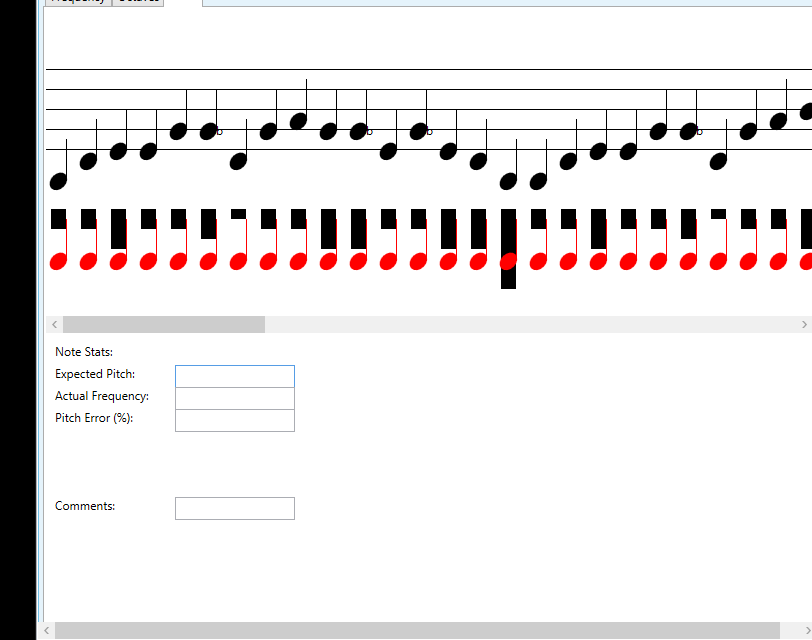
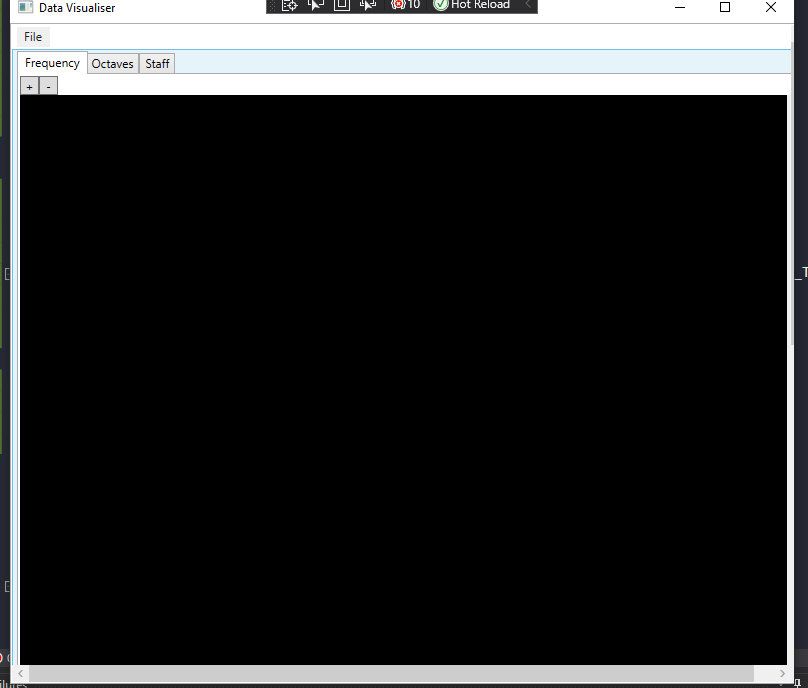


Figure 20(wrong output)

# Explanation Code

## timefreq.cs

Add variables and changing some variables to global use

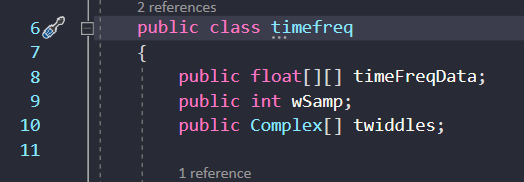


Figure 21(sequential timefreq)

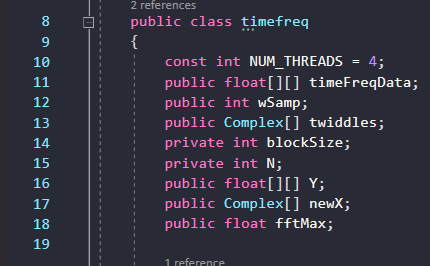


Figure 22(parallel timefreq)

First loop parallel

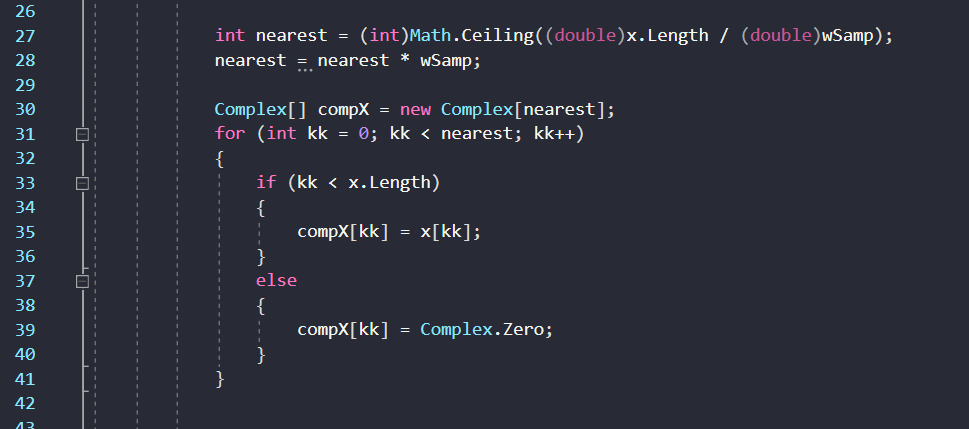


Figure 23(sequential timefreq)

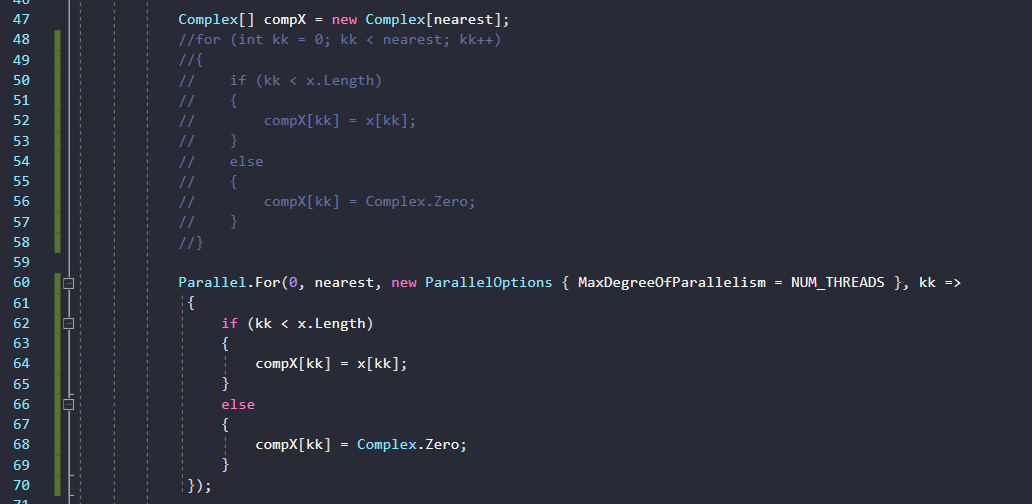


Figure 24(parallel timefreq)

Second loop parallel

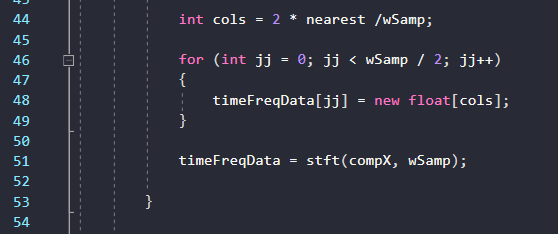


Figure 25(sequential timefreq)

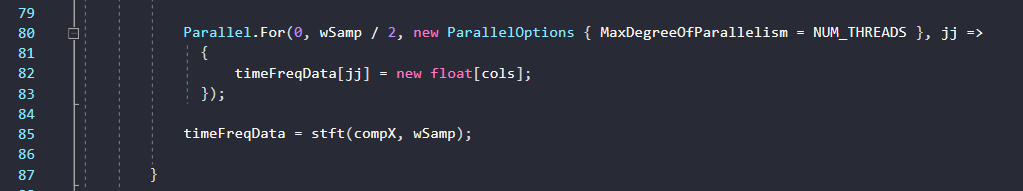


Figure 26(parallel timefreq)

First loop in side stft

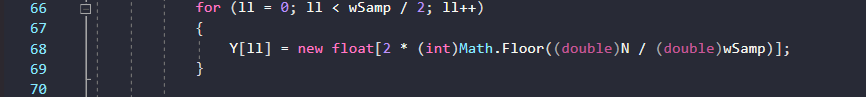


Figure 27(sequential stft)

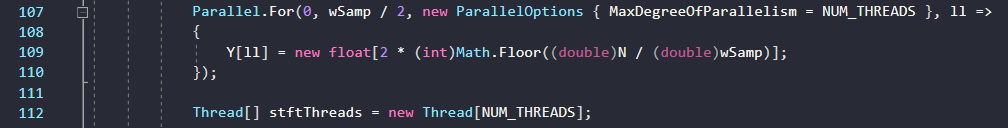


Figure 28(parallel stft)

Second loop in stft

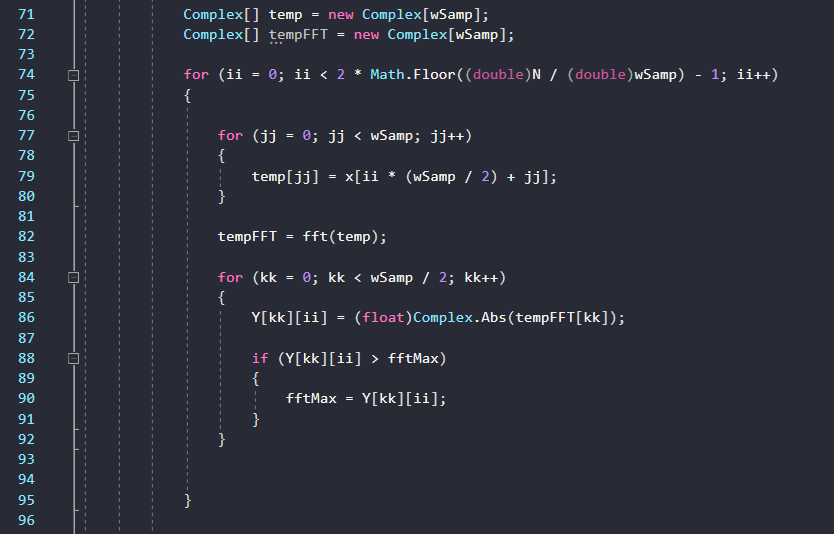


Figure 29(sequential stft)

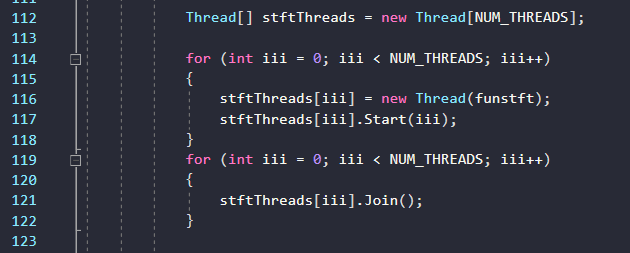


Figure 30(paralleled stft 1)

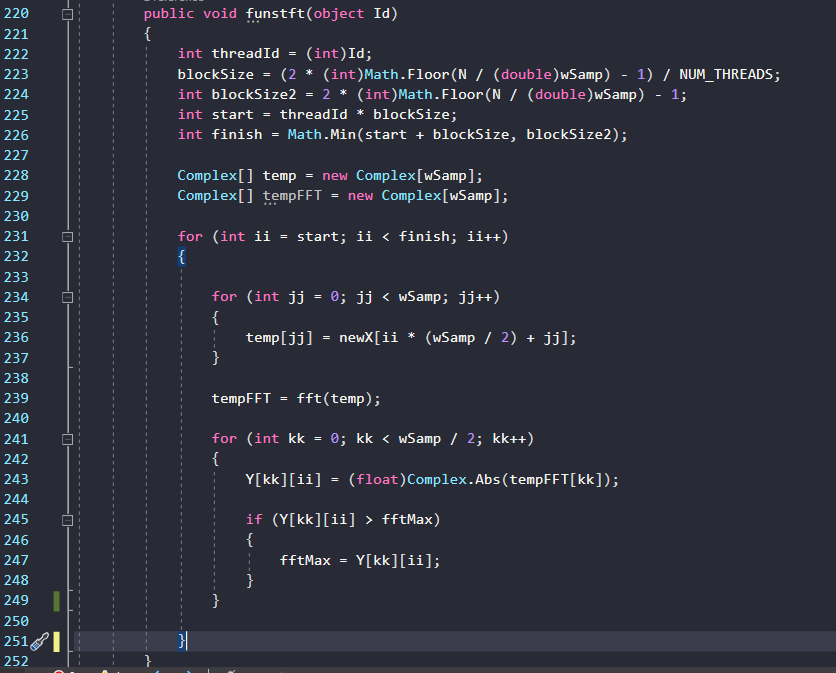


Figure 31(paralleled stft 2)

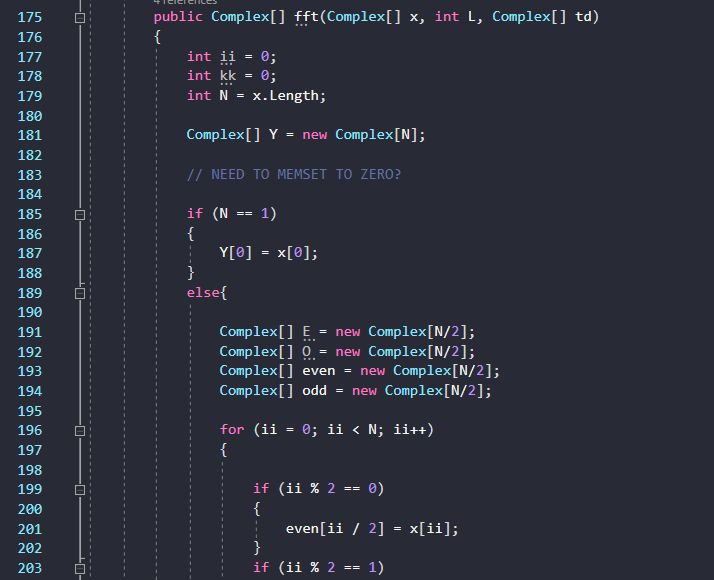


Figure 32(changed FFT's parameters)

## mainwindow.cs

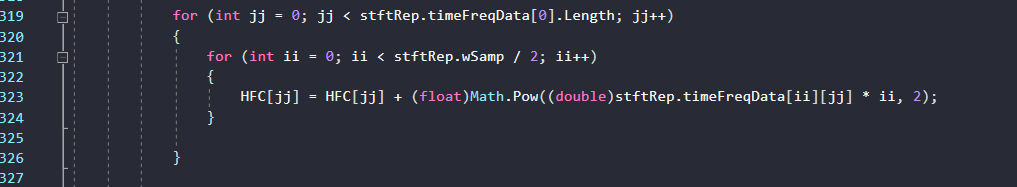


Figure 33(sequential mainwindow)

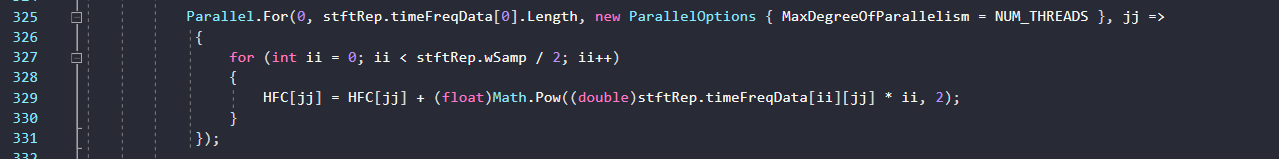


Figure 34(paralleled mainwindow)

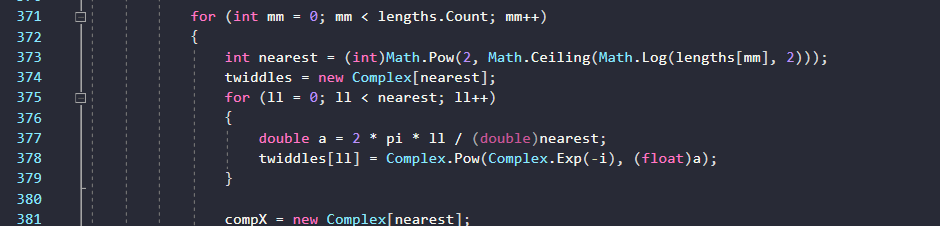


Figure 35(sequential mainwindow)

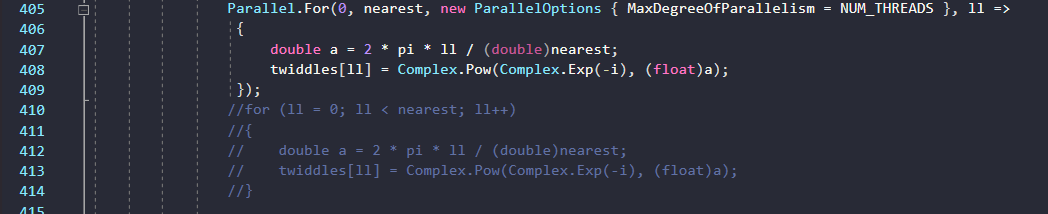


Figure 36(paralleled mainwindow)

# Reflection

After completing this project, several goals could be archived, but due to the amount of knowledge and limited time I have, I could not achieve those goals. The first goal for me that I failed to archive is paralleling FFT. The entire application’s calculation is mostly done by the method FFT, by transforming the algorithm or creating a new algorithm will significantly reduce the time it takes. I believe to parallel FFT fully, it will require the algorithm to be transformed. In addition, I also believe that there are still ways to parallel the onsetdetection function. The parallel version of my onsetdetection is very basic, and it can still be worked on, including finding more places to parallel, transform or create new algorithms.

Furthermore, After doing some research, the parallelism of this application can be done by using workers from the BackgroundWoker class. However, I did not spend time researching the worker method until the last week. Lastly, the final method that I didn’t try out is using my GPU. I’m currently using a NIVIDA GPU, and NVIDIA GPU provides users with a CUDA library, which is a library that helps users to parallel applications.

Overall, after completing this project, my perceptive on programming has changed a lot. I always thought that programming is simply writing code that can run and achieve your goals without worrying how long it will take to run. This project has taught me that writing code is easy, but making it run as efficiently as possible, is very complicated. Most of the time, there is no end to it. There is always a way around it to make it faster, including new algorithms, transform algorithms, threadings or even new hardware. In the future, I can see myself trying out new ways to parallel applications and changing the approach on how to start writing a program with parallelisation in my head.