CAB401 High Performance and Parallel Computing

Project Assignment
(DigitalMusicAnalysis)

Queensland University of Technology

Pattarachai Roongsritong n10548467

Table of Contents

1. Introduction

This report is going to demonstrate an attempt to enhance the performance of a program called "digitalMusicAnalysis" which is a program to help rookie violin players get their performance feedback in terms of correctness. This project aims to achieve through leveraging parallel computing tools and techniques.

1.1 Digital Music Analysis program overview

As mentioned earlier, the program is used to give a violin player feedback of his/her recorded performance whether the high (sharp) or the low (flat) notes are played correctly. The program takes a WAV file input in companion with an XML file input as a reference then visualises sound frequencies and octaves highlighting the difference from correct octaves according to the reference. The UML class diagram is shown in the **figure 1**.

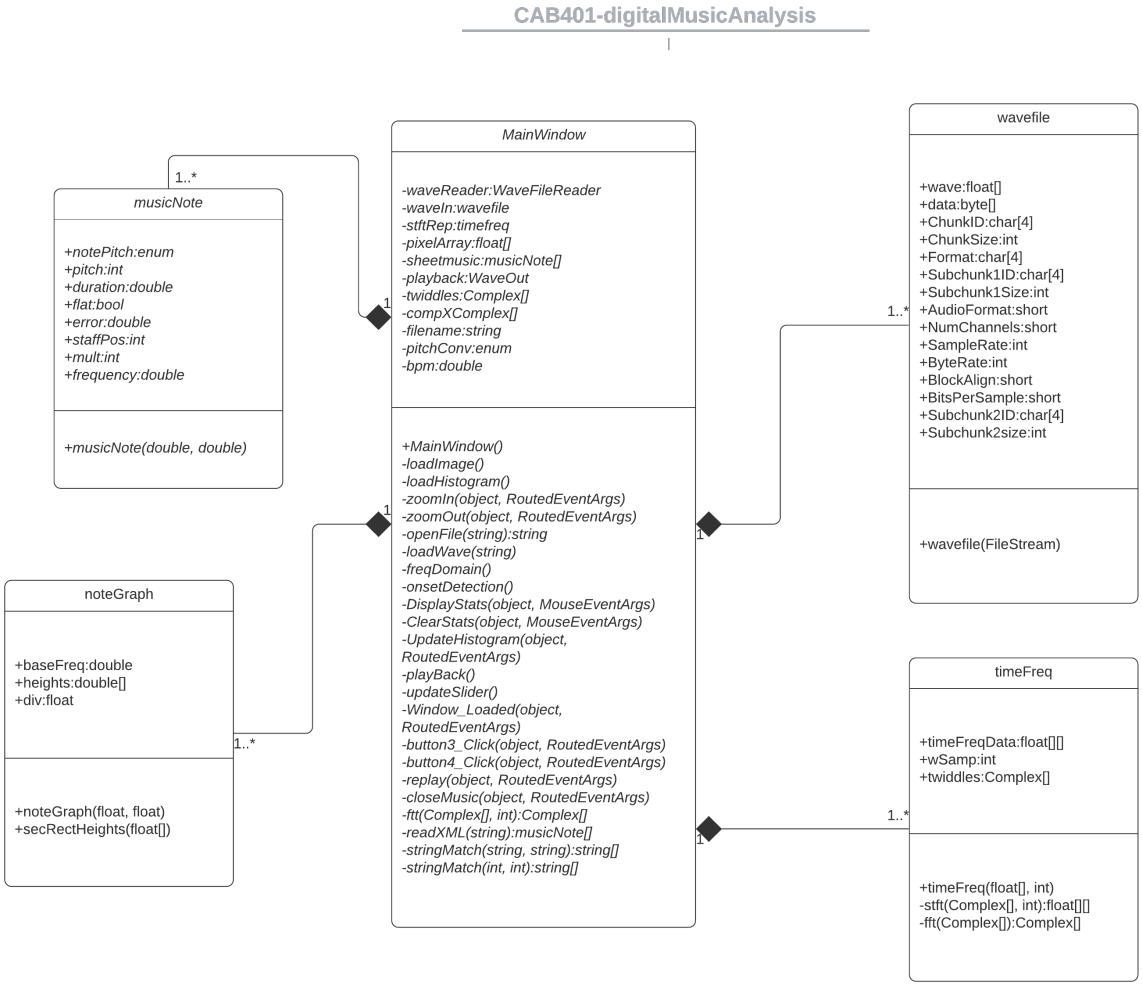


Fig.1 UML class diagram of the digitalMusicAnalysis program

2. Accessed Tools

2.1 Hardware

Since I can only access to a MAC, The QUT virtual machine is used via the VMware Horizontal Client to parallelise the application which has the following specifications.
Processor - Intel Xeon Processor E5-2687W v4
Clock speed – 3.00 GHz
Cache – 30MB
The number of cores – 12
The number of threads – 24
Installed memory – 16.0 GB

2.2 Software

The original program was developed in C# thus the Microsoft Visual Studio 2019 is used to achieve parallelism by the Task Parallel Library (TPL) as well as the built-in performance profiler to analyse the CPU usage data. Moreover, the built-in C# library is used to track the running time.

3. Parallelising Implementation

3.1 Preliminary profiling

The performance profiler is used to determine how much CPU resources are spent in each program component. According to the result as shown in the **figure 2,** it suggests that the fff functions spend the most CPU resources. The functions appear in two spots of the program which are the MainWindow class and the timefreq class. Moreover, the hot path reveals that the two fff functions are invoked through the freqDomain (line 48) and the onsetDetection (line 51) functions. Subsequently, when taking a closer look at the freqDomain function, the function spends most of its time instantiating an object from timeFreq class (DigitalMusicAnalysis.timefreq..ctor(float[]32, int)). As a result, our main investigating spots are the timeFreq class and the onsetDetection function.

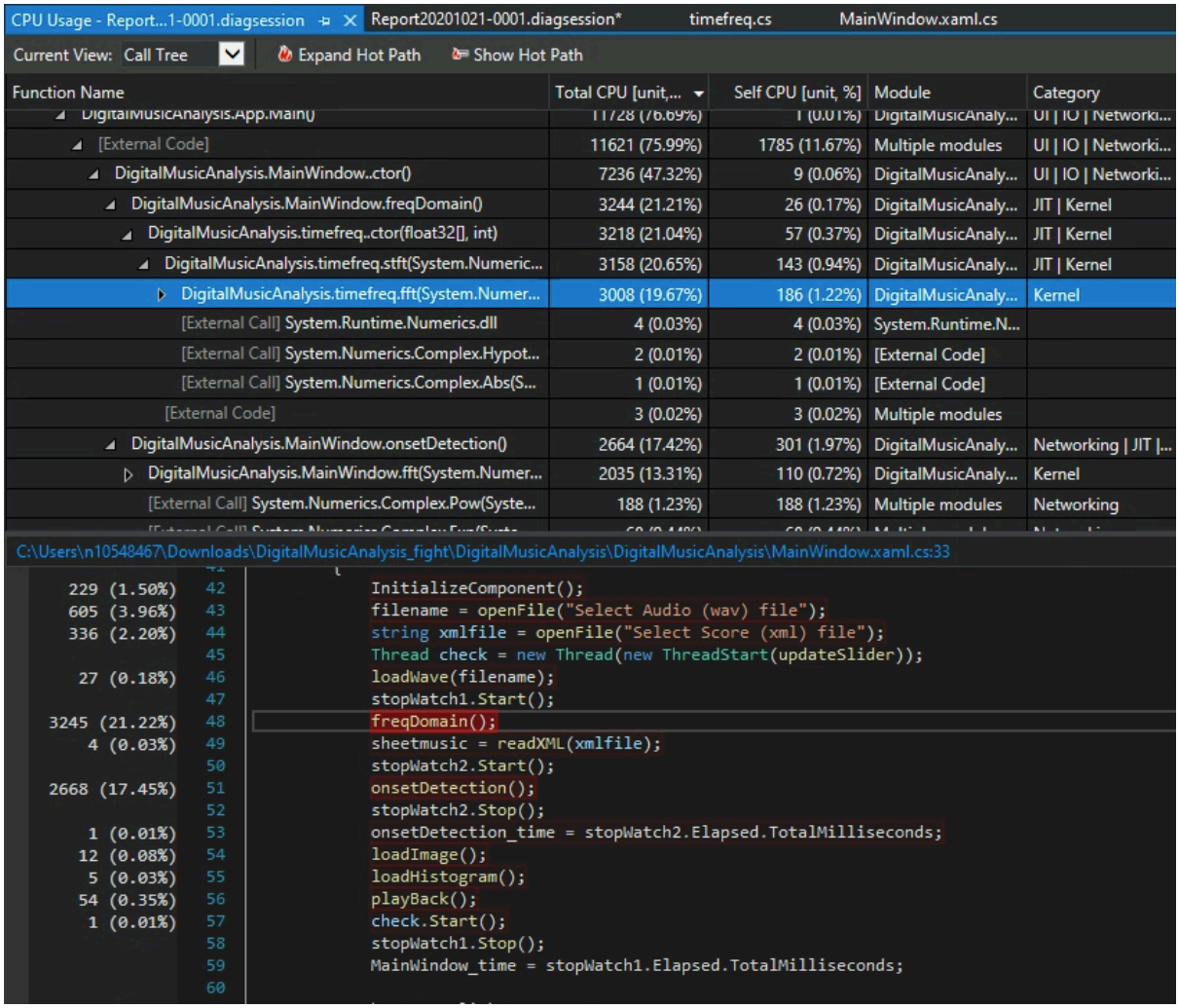


Fig.2 Profiling result of the sequential version program

3.2 timefreq class

Initial analysis

According to the profiler shown in **figure 3, 4, and 5**, within the constructor of this class (line 285 of MainWindow.xml.cs), the main work is done by calling the stff function (line 53 of timefreq.cs) to calculate a two-dimensional array of floating-point numbers which requires to call the fff function (line 87 of timefreq.cs) a number of times. In order to safely parallelise, there are two data dependencies to be considered which are the two temporary arrays of complex numbers (line 76 and 77 of timefreq.cs). Additionally, the profiler also suggests that the loop involving with a calculation of twiddle numbers **(figure 6)** and the loop calculating the Y array **(figure 7)** have some sample counts, so they are interesting to be parallelised as well because they carry no dependency.

C:\Users\n10548467\Downloads\DigitalMusicAnalysis_fight\DigitalMusicAnalysis\DigitalMusicAnalysis\MainWindow.xaml.cs:284

```
278
279         }
280
281         // Transforms data into Time-Frequency representation
282
283         private void freqDomain()
284         {
3219 (21.05%) 285             stftRep = new timefreq(waveIn.wave, 2048);
286             pixelArray = new float[stftRep.timeFreqData[0].Length * stftRep.wSamp / 2];
287             for (int jj = 0; jj < stftRep.wSamp / 2; jj++)
288             {
8 (0.05%) 289                 for (int ii = 0; ii < stftRep.timeFreqData[0].Length; ii++)
1 (0.01%) 290                 {
11 (0.07%) 291                     pixelArray[jj * stftRep.timeFreqData[0].Length + ii] = stftRep.timeFreqData[jj][ii];
5 (0.03%) 292                 }
293             }
294
295         }
296
```

Fig.3 Using timefreq constructor in freqDomain()

C:\Users\n10548467\Downloads\DigitalMusicAnalysis_fight\DigitalMusicAnalysis\DigitalMusicAnalysis\timefreq.cs:12

```
45
46             int cols = 2 * nearest /wSamp;
47
48             for (int jj = 0; jj < wSamp / 2; jj++)
49             {
9 (0.06%) 50                 timeFreqData[jj] = new float[cols];
51             }
52             stopWatch.Start();
3159 (20.66%) 53             timeFreqData = stft(compX, wSamp);
54             stopWatch.Stop();
1 (0.01%) 55             stft_time = stopWatch.Elapsed.TotalMilliseconds;
56         }
57
58         float[][] stft(Complex[] x, int wSamp)
59         {
60             int ii = 0;
61             int jj = 0;
62             int kk = 0;
63             int ll = 0;
```

Fig.4 Invoking of stft function

```
58         float[][] stft(Complex[] x, int wSamp)
59         {
60             int ii = 0;
61             int jj = 0;
62             int kk = 0;
63             int ll = 0;
64             int N = x.Length;
65             float fftMax = 0;
66
67             float[][] Y = new float[wSamp / 2][];
68
69
70
71             for (ll = 0; ll < wSamp / 2; ll++)
72             {
13 (0.09%) 73                 Y[ll] = new float[2 * (int)Math.Floor((double)N / (double)wSamp)];
74             }
75
76             Complex[] temp = new Complex[wSamp];
77             Complex[] tempFFT = new Complex[wSamp];
78
79             for (ii = 0; ii < 2 * Math.Floor((double)N / (double)wSamp) - 1; ii++)
80             {
81
5 (0.03%) 82                 for (jj = 0; jj < wSamp; jj++)
1 (0.01%) 83                 {
27 (0.18%) 84                     temp[jj] = x[ii * (wSamp / 2) + jj];
3 (0.02%) 85                 }
86
3009 (19.68%) 87                 tempFFT = fft(temp);
88
7 (0.05%) 89                 for (kk = 0; kk < wSamp / 2; kk++)
90                 {
44 (0.29%) 91                     Y[kk][ii] = (float)Complex.Abs(tempFFT[kk]);
92
20 (0.13%) 93                     if (Y[kk][ii] > fftMax)
94                     {
95                         fftMax = Y[kk][ii];
96                     }
1 (0.01%) 97                 }
98
```

Fig.5 Invoking fft function from stft function

```
20          twiddles = new Complex[wSamp];
21          for (ii = 0; ii < wSamp; ii++)
22          {
23              double a = 2 * pi * ii / (double)wSamp;
24              twiddles[ii] = Complex.Pow(Complex.Exp(-i), (float)a);
25          }
26
```

Fig.6 A loop of Twiddle numbers calculation

```
70
71          for (ll = 0; ll < wSamp / 2; ll++)
72          {
14 (0.12%)  73              Y[ll] = new float[2 * (int)Math.Floor((double)N / (double)wSamp)];
74          }
75
```

Fig.7 A loop of Y array calculation

Approach

Due to its dependencies, the whole code-block of ii loop containing the fff function is performed on its own thread to achieve parallelism by creating a new method as the **figure 8** called stftOnThread to run on an individual thread as the **figure 9**. The loops for calculating twiddle values and Y array are also implemented parallelisation by the Parallel.For method which the iterations are able to run in parallel and the maximum number of parallel tasks can be set according to our needs as shown in **figure 10 and 11**.

C:\Users\n10548467\Downloads\DigitalMusicAnalysis_fight\DigitalMusicAnalysis - Parallelised\DigitalMusicAnalysis\timefreq.cs:25

```
118         public void stftOnThread(object threadID)
119         {
120             int id = (int)threadID;
121             int start = id * chunk_size;
122
123             Complex[] temp = new Complex[wSamp];
124             Complex[] tempFFT = new Complex[wSamp];
125
126             for (int ii = start; ii < Math.Min(start + chunk_size, array_length - 1); ii++)
127             {
128                 for (int jj = 0; jj < wSamp; jj++)
129                 {
130                     temp[jj] = newX[ii * (wSamp / 2) + jj];
131                 }
132
133                 tempFFT = fft(temp);
134
135                 for (int kk = 0; kk < wSamp / 2; kk++)
136                 {
137                     Y[kk][ii] = (float)Complex.Abs(tempFFT[kk]);
138
139                     if (Y[kk][ii] > fftMax)
140                     {
141                         fftMax = Y[kk][ii];
142                     }
143                 }
144             }
145
```

Fig.8 stftOnThread Method

```
73          float[][] stft(Complex[] x, int wSamp)
74          {
75              N = x.Length;
76              fftMax = 0;
77              newX = x;
78              array_length = 2 * (int)Math.Floor(N / (double)wSamp)-1;
79              chunk_size = (array_length + MainWindow.NUM_THREADS - 1) / MainWindow.NUM_THREADS;
80
81              Y = new float[wSamp / 2][];
82
83
84
85              Parallel.For(0, wSamp / 2, new ParallelOptions { MaxDegreeOfParallelism = MainWindow.NUM_THREADS }, ll =>
86              {
87                  Y[ll] = new float[2 * (int)Math.Floor((double)N / (double)wSamp)];
88              });
89
90
91
92              Thread[] fft_thread = new Thread[MainWindow.NUM_THREADS];
93
94              for (int i = 0; i < MainWindow.NUM_THREADS; i++)
95              {
96                  fft_thread[i] = new Thread(stftOnThread);
97                  fft_thread[i].Start(i);
98              }
99              for (int j = 0; j < MainWindow.NUM_THREADS; j++)
100             {
101                 fft_thread[j].Join();
102             }
103
104
105
106             for (int ii = 0; ii < 2 * Math.Floor((double)N / (double)wSamp) - 1; ii++)
107             {
108                 for (int kk = 0; kk < wSamp / 2; kk++)
109                 {
110                     Y[kk][ii] /= fftMax;
111                 }
112             }
113
```

Fig.9 Parallelising stft function by implementing stftOnThread

```
34              twiddles = new Complex[wSamp];
18 (0.10%)      Parallel.For(0, wSamp, MainWindow.parallel_options, ii =>
36                  {
37                      double a = 2 * pi * ii / (double)wSamp;
38                      twiddles[ii] = Complex.Pow(Complex.Exp(-i), (float)a);
39                  });
40
```

Fig.10 Parallelising a loop of Twiddle numbers calculation

```
84
85          Parallel.For(0, wSamp / 2, new ParallelOptions { MaxDegreeOfParallelism = MainWindow.NUM_THREADS }, ll =>
86          {
87              Y[ll] = new float[2 * (int)Math.Floor((double)N / (double)wSamp)];
88          });
89
```

Fig.11 Parallelising a loop of Y array calculation

3.3 onsetDetection method

Initial analysis

As discussed in the preliminary profiling section, this method involves in calling the fff function multiple times and when we take a look at the code block, it also has its own expensive calculations which can be listed as follows:

- HFC array calculation loop

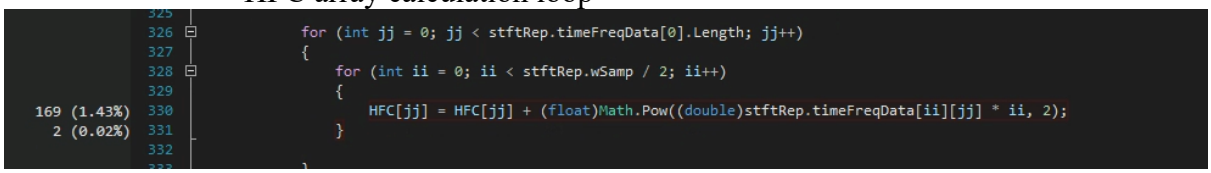


Fig.12 HFC array calculation loop (sequential)

- twiddles array calculation loop
- complex number array calculation loop which is an argument for the fff function

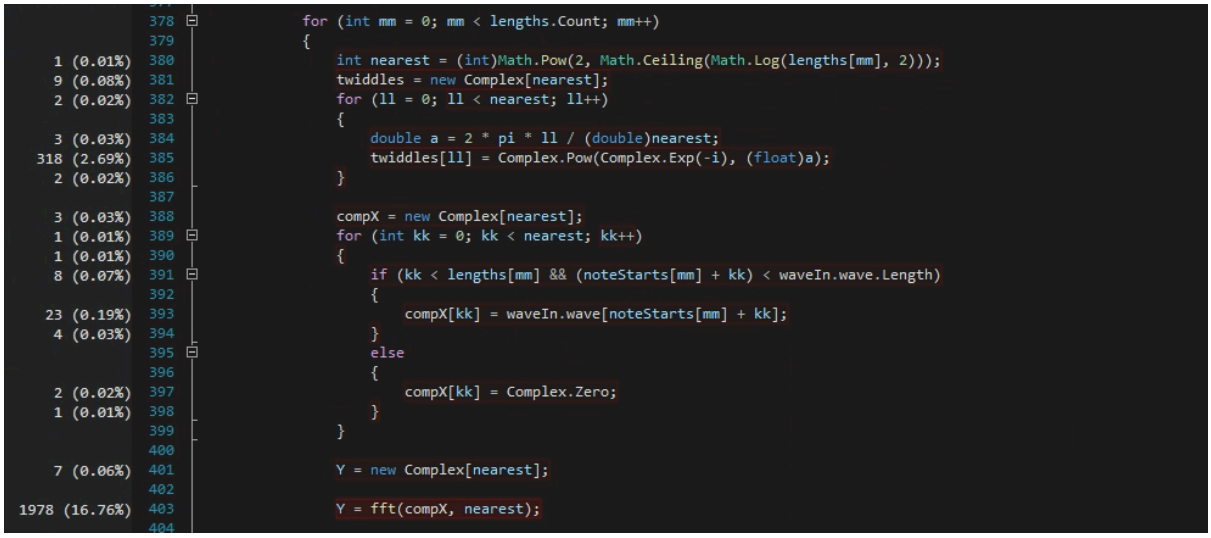


Fig.13 Twiddles array and complex number array calculation loops

- absolute values of Y array calculation loop

In order to enhance the performance of this method, the aforementioned loops will be given parallelism attempts.

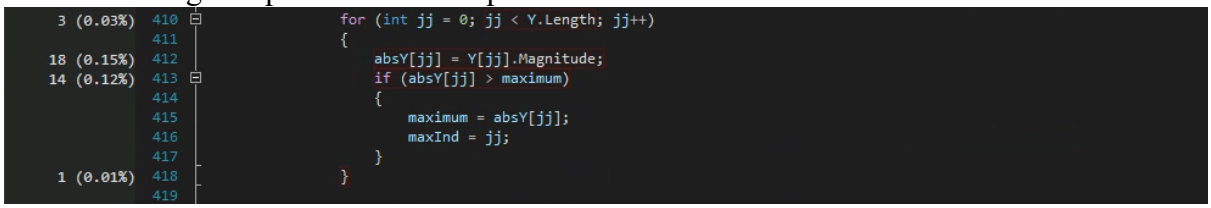


Fig.14 Absolute values of Y array calculation loop

Approach

- HFC array calculation loop: It seems that the structure of the loop allows temporal locality because the outer loop index jj is used to read and write the HFC array and read the timeFreqData array same elements multiple times within the inner loop ii which is likely to result in cache hit without a loop transformation; however, the HFC array calculation loop carries no

dependency so it is safe to parallelise which can potentially save some computing resources.

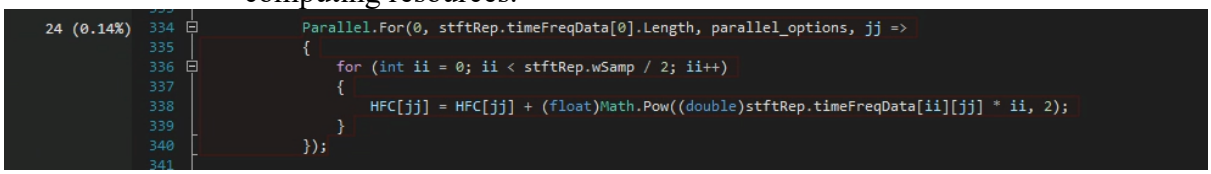


Fig.15 Parallelised HFC loop

- twiddles array and complex number array calculation loops: These two calculations can be parallelised by using a loop fusion since both loops keep running until the index is less than the variable called nearest and still can preserve the same data dependency (the variable a) and control dependency (if-else statement). Unfortunately, the attempt to parallelise the fff function calling is fail despite having done different ways of restructuring at the mm loop because its arguments (compX and nearest) carry data dependencies.

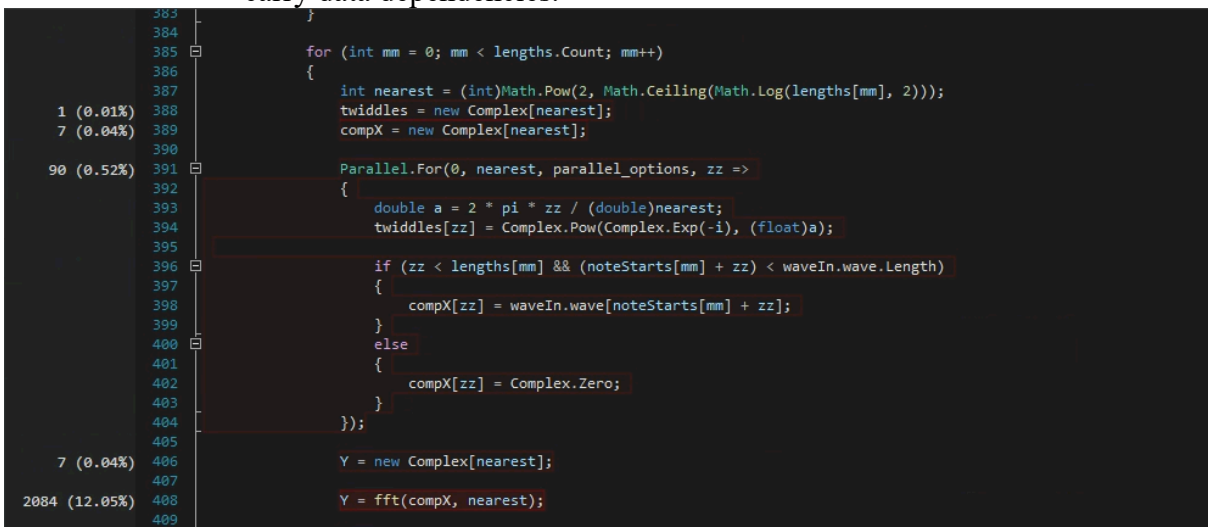


Fig.16 Parallelised Twiddles array and complex number array calculation loops

- absolute values of Y array calculation loop: The operation which has a certain sample counts in this loop is assigning absolute values of Y array, so the loop is transformed by a loop distribution and parallelised.

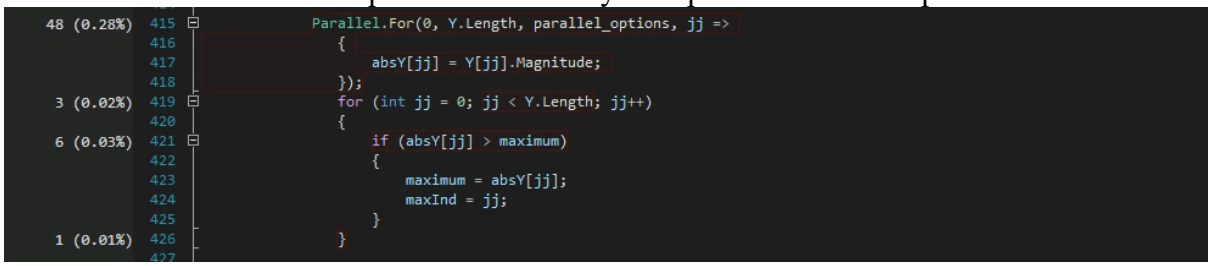


Fig.17 Parallelised absolute values of Y array calculation loop

4. Results

The Stopwatch class is used to measure the time used to execute three functions including MainWindow representing total runtime of the program, stft, and onsetDetection program which the latter two are our hot spot to attempt parallelisation.

4.1 Sequential version

The time taken of each function can be summarised in the **table 1**.

Method name	**time (ms)**
MainWindow	3642.3662
stft	1754.5105
onsetDetection	1618.3864

Table 1. Spent time on sequential version

4.2 Parallelised version

In the **figure 18,** it can be seen that a profiling result has changed from the sequential version as the freqDomain function does not spend as much CPU resources as in prior profiling result. However, the onsetDetection function still consumes significant CPU resources.

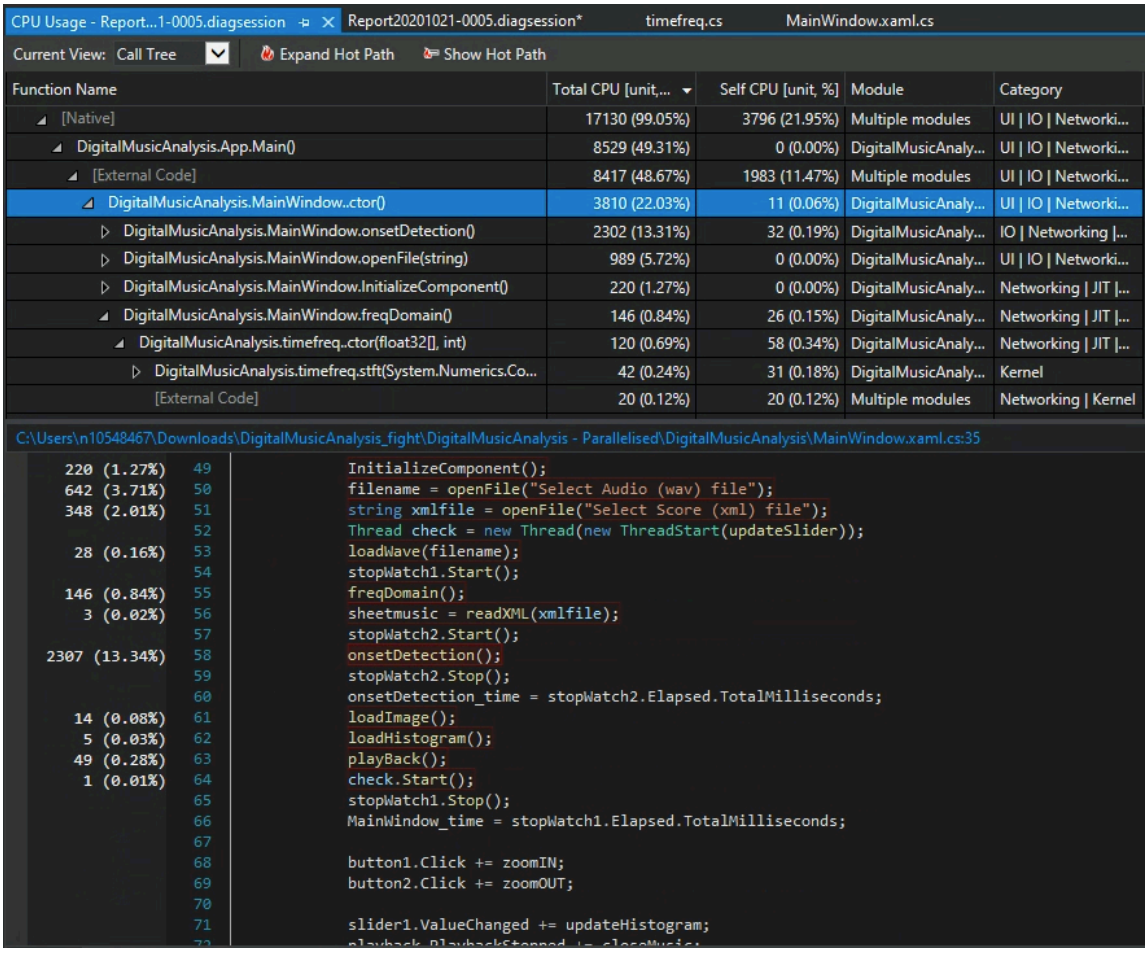


Fig.18 Profiling result of the parallelised version of the application

The time used in running each function is recorded in different level of parallelism exposure ranging from 1 to 8 processor counts. Moreover, a speedup ratio is calculated for each processor count of the MainWindow method as demonstrated in **table 2**.

MainWindow			stff		onsetDetection	
Processor count	**time (ms)**	**speed-up**	**Processor count**	**time (ms)**	**Processor count**	**time (ms)**
1	3272.7317	1.112943722	1	1484.1659	1	1540.8411
2	2602.9121	1.399342759	2	962.7147	2	1360.9276
3	2542.3967	1.432650617	3	855.7366	3	1388.0724
4	2425.2537	1.501849559	4	783.8631	4	1325.9001
5	2212.5913	1.646199278	5	698.0564	5	1208.3032
6	2190.7844	1.662585419	6	747.5258	6	1150.488
7	2363.2993	1.541220869	7	749.6073	7	1226.7276
8	2334.0133	1.560559316	8	814.3922	8	1208.1522

Table 1. Spent time on parallel version

After the speedup ratios are obtained, a speedup curve can be plotted as in the **figure 19**. It appears that the parallel version of the application can achieve slightly speedup

in a sublinear manner. This probably because despite having done a great job on the stff function, it is failed to enhance the performance of the onsetDetection function. Additionally, the stff seems still not scalable because after hit 5 processors, the time taken is stagnant and then increased.

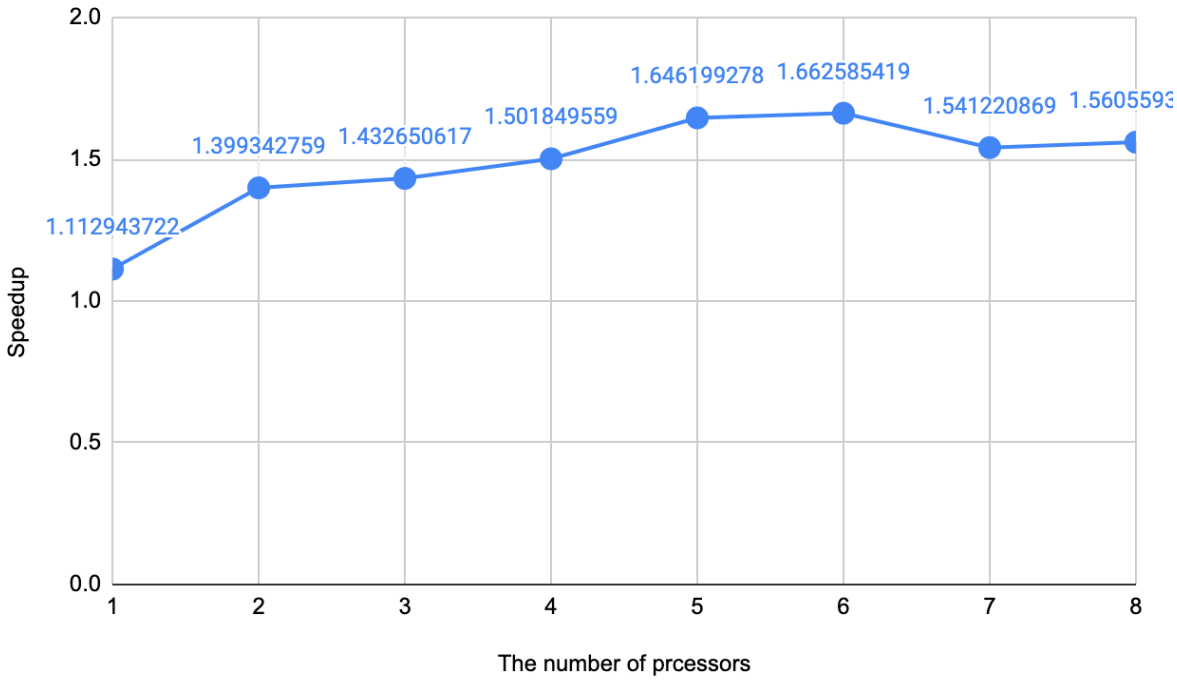


Fig.19 A speedup curve

4.3 Outputs

Since the outputs of the application are visualised by its nature, we can visually compare the outputs from the frequency and staff tabs of the program as in .

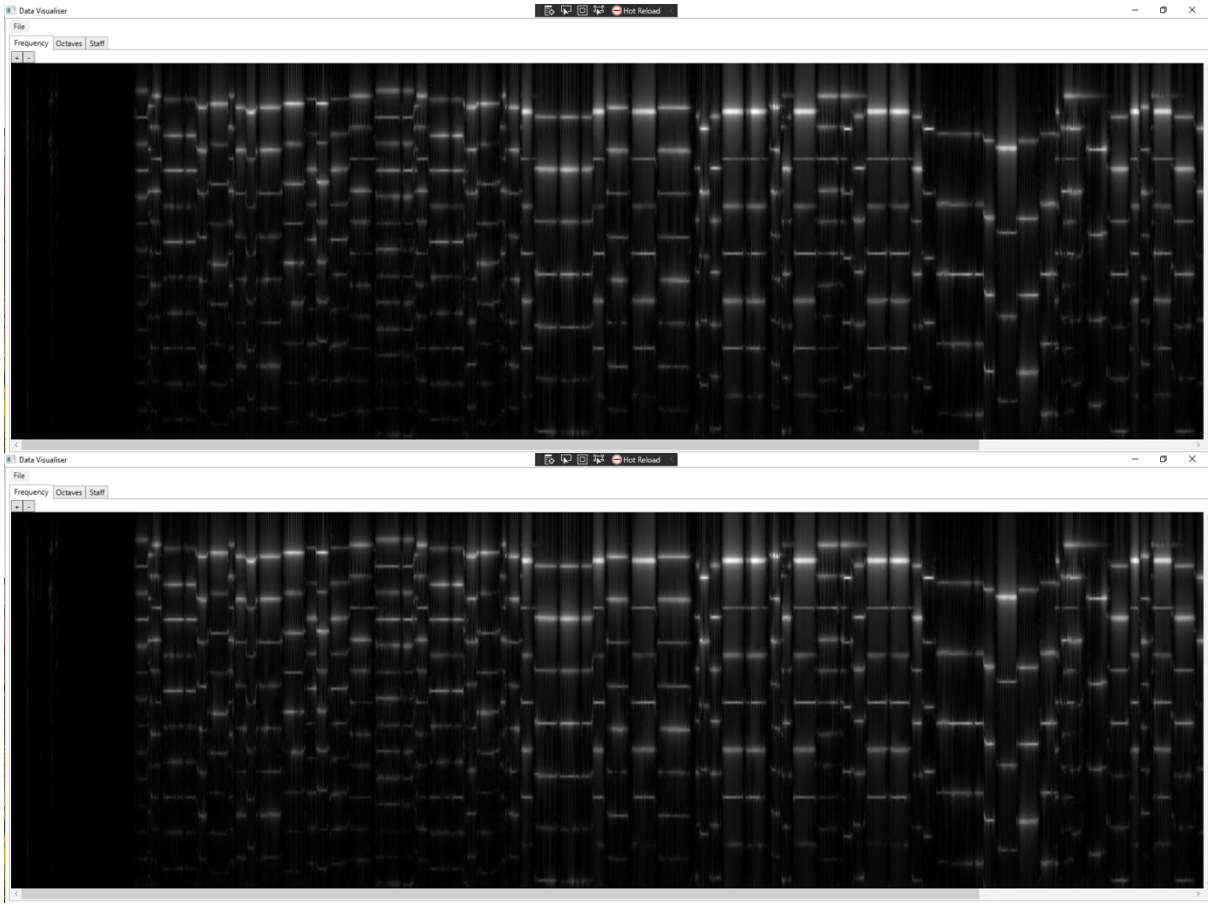
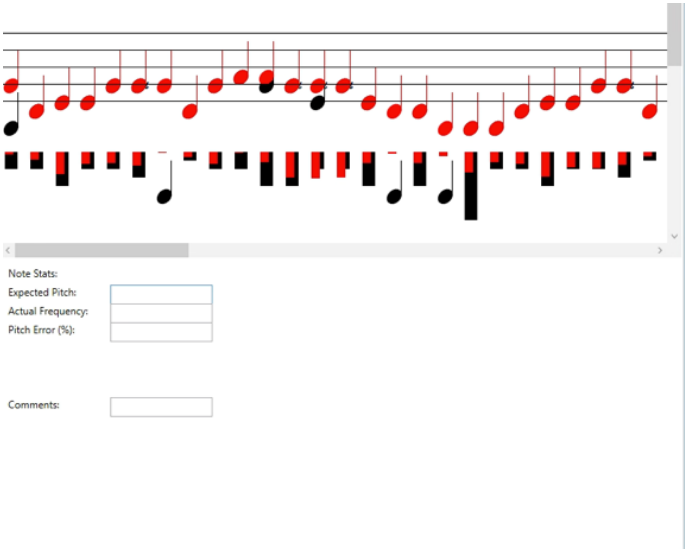


Fig. 20 Outputs of Frequency tab (upper: sequential version, lower: parallel version)



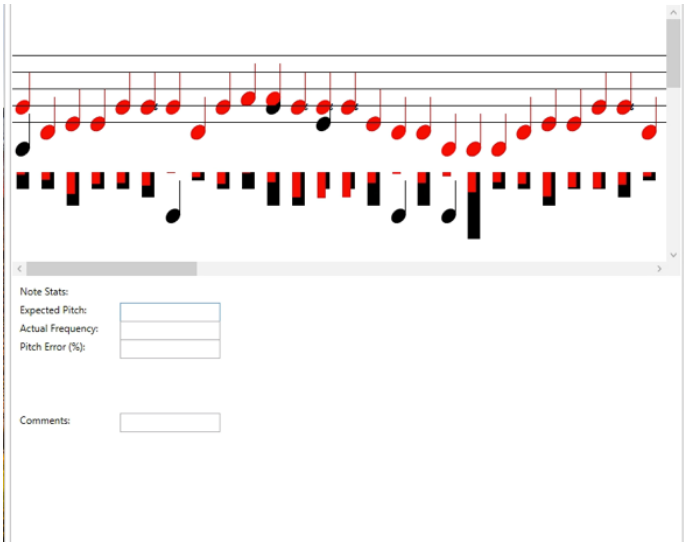


Fig. 21 Outputs of Staff tab (upper: sequential version, lower: parallel version)

5. Reflection

In this parallelisation assignment, I had an opportunity to enhance the performance of a substantial program application. Having look at the source-code at first, it was intimidating to see a number of lines and methods. Fortunately, there was a systematic approach to start solving a problem like this through using the Microsoft Visual Studio Profiler which I have never used before to grasp where the resource being spent substantially in specific sections of the program. This makes me think it will be very useful in my future career too. Subsequently, I was exposed to different ways to analyse variable dependencies as well as a number of ways to restructure a program to safely boost its performance from the lower-level abstraction instead of merely parallelise a loop. Above all, I was able to safely improve the performance of stff function by running in separated threads. However, I failed to increase speedup for the onsetDetection function as the dependencies could not be preserve so it results in slightly different in terms of execution time making the overall run time does not much improve. Finally, the future work is potentially changing from the current recursive FFT algorithm to be in an iterative form which can calculate each sample output.

6. Appendices

6.1 Appendix A

- Microsoft Task Parallel Library (TPL) https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl
- Parallel.For method https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallel.for?view=netcore-3.1
- The maximum number of concurrent tasks

https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.paralleloptions.maxdegreeofparallelism?view=netcore-3.1

- Stopwatch class
  https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallel.for?view=netcore-3.1

6.2 Appendix B

Console outputs describing the time taken in each function