2019

CAB401: High Performance and Parallel Computing

DIGITAL MUSIC ANALYSIS

ARIK INTENAM MIR, N9637567

Contents

1.0 Introduction

The application that was chosen for the parallelizing and optimizing runtime speed is called Digital Music Analysis. This is an object -oriented application with interfaces, written using the programming language C#. The application determines the highs and lows of a given audio sample by analyzing the frequencies of the notes, that are being played on the background, and finally displays a frequency visualization of the audio samples along with the octaves of played notes. The sequential application was parallelized using the Task Parallel Library (TPL).

2.0 Tools for Parallelizing

2.1 Hardware:

- **Device:** ASUS VivoBook S
- **Processor:** Intel(R) Core (TM) i7-8550 CPU
- **Clock speed:** 1.80 GHz
- **Cores:** 4
- **Logical Processors:** 8
- **Virtualization:** Enabled
- **L1 Cache:** 256KB
- **L2 Cache:** 1.0MB
- **L3 Cache:** 8.0MB
- **Memory:** 32GB DDR4

2.2 Software:

- **Development Framework:** Microsoft Visual Studio 2019
- **Language:** C#
- **Parallelization Framework:** Task Parallel Library (TPL)
- **Performance Analyzer:** Visual Studio Performance Profiler

3.0 Application Overview: Sequential implementation

A screenshot of the sequential application's interface is attached in **APPENDIX A.**

- The sequential Digital Music Analysis application initiates by taking in an audio file in the .wav format and an XML reference file in the .xml format. Initially, the .wav file is compared with the .xml for marking out the errors in the played note.
- Then the method *freqDomain*() from class MainWindow.xaml.cs will analyze and change the digital signal into frequency arrays of type float to 2048 samples thus converting the signal from time domain to its frequency domain.
- The created float arrays of frequency will be compared with the other read-in musical notation .xml file.
- Most of the work for the frequency domain are performed in the timefreq Class, where the most significant methods are the *stft*() and *fft*(). The float arrays generated by the *freqDomain*() method will be used in the timefreq class as input to generate a complex array.

- *stft*() or short for “Short-term Fourier transform” method will take the complex array as input and will return a single two-dimensional array of type float. Where, the two dimensions consists of time access and another one for the frequency. *stft*() also helps to understand and visualize the magnitude of the frequency in the given 2048 samples.
- The most significant method of the whole sequential application is the *fft*(). It is implemented in both MainWindow.xaml.cs and timefreq.cs. *fft* is short for “Fast Fourier Transform”. *fft* was implemented here using the Cooley-Tukey Algorithm in a recursive manner. Cooley-Tukey algorithm is a good example of the divide and conquer rule. The input complex array is further divided to two new arrays containing the even and odd elements.

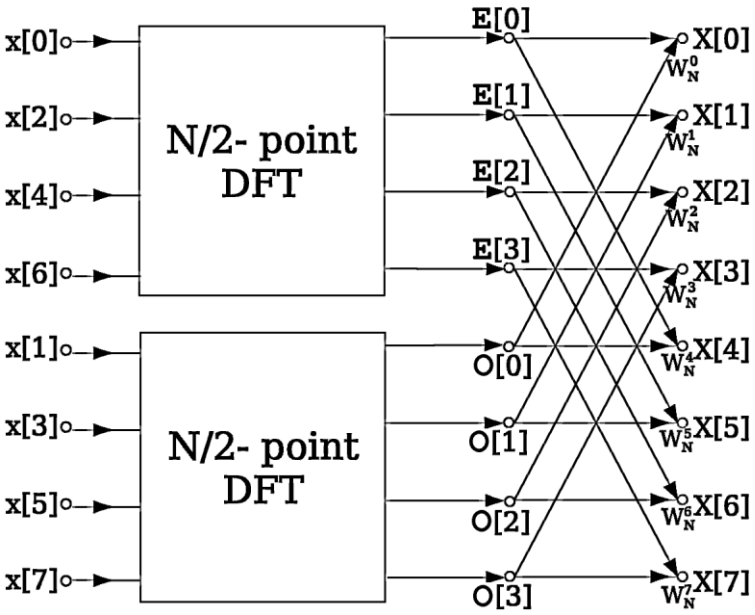

Figure 1: Implementation of FFT Cooley-Tukey algorithm

To better understand how the whole application works, a reference has been made to the **APPENDIX B,** where an UML class diagram and a class diagram generated by Visual Studio is attached.

3.1 Analysis of the Sequential application:

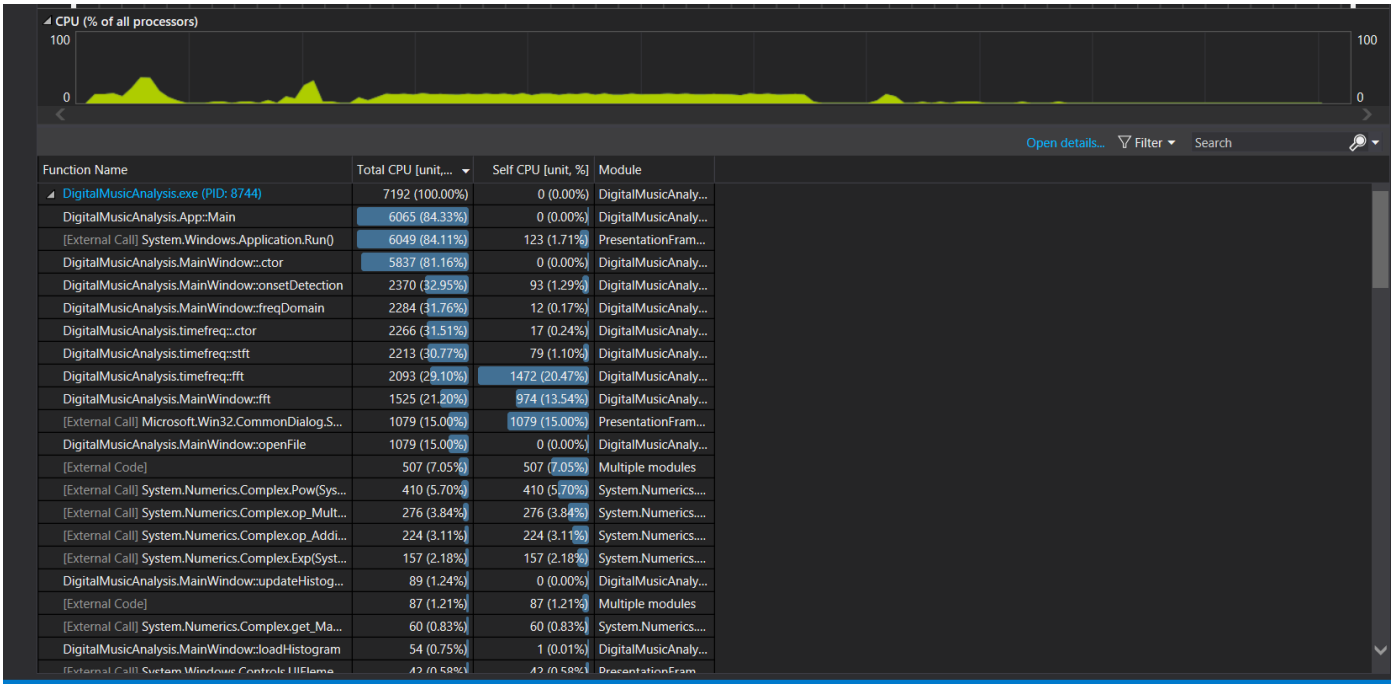

Figure 2: CPU usage for the Sequential application

Performance profiler from the Visual studio shows that the application spends a huge amount of time processing the data in the *fft*() methods. I believe the reason for this is the recursive manner it was implemented in, which also limited the scope of parallelism as it was also called in more than one places. Another method worthy of attention is the *onsetDetection*() method from MainWindow.xaml.cs. Its interesting that it calls the *fft* function as well as does a lot of its own calculations, thus making it ideal for parallelizing. The initial function call graph also supports the opinion that *onsetDetection*() and *fft*() methods uses the highest amount of resources, the function call graph is attached in the **APPENDIX C.** The non-significant method calls were excluded from the call graph, as they do not use up much resources from the processors. From the call graph, it is observed that the constructor method of the MainWindow.xaml.cs has the highest percentage. Though, I think there are not much scope of parallelizing in this method, as it spends significant amount of resources in calling other functions. Lastly, the function *freqDomain*() from the MainWindow.xaml.cs is also note-worthy as it performs the task of transforming digital signals into the frequency domain. So, I believe there will be scope of parallelizing this method too.

4.0 Implementation of Parallelization

In order to make the application operate faster, numerous parallelization attempts were taken. These attempts involved using TPL to run faster, reconstruction of algorithms and splitting large loops into smaller parts for exposing potential scopes for parallelization.

4.1 Parallelizing timefreq Class

Initially I assumed that timefreq class spends a lot of time processing. Then after profiling the class it was proven to me that the timefreq class was spending a lot of time computing the twiddles value. Along with the constructor, there were two other methods in this class, namely *stft*() and *fft*().

4.1.1 Parallelizing *timefreq()* Constructor and *stft()*

As the *timefreq* constructor was using a lot of processor resource for calculating the twiddles value, I achieved improve in performance in the constructor by implementing a Parallel.For loop for computing the twiddles. Achieving optimization here was easy as there were no dependencies present in the loop.

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

timeFreqData = new float[wSamp/2][];
```

Figure 3: Before parallelizing timefreq constructor

```
1 reference
public timefreq(float[] x, int windowSamp)
{
    double pi = 3.14159265;
    Complex i = Complex.ImaginaryOne;
    this.wSamp = windowSamp;
    twiddles = new Complex[wSamp];

    Parallel.For(0, (wSamp), (int ii, ParallelLoopState state) =>
    {
        double a = 2 * pi * ii / (double)wSamp;
        twiddles[ii] = Complex.Pow(Complex.Exp(-i), (float)a);
    });
```

Figure 4: After parallelizing timefreq constructor

And for optimizing the *stft*() method, I tried to run each call of the *fft*() in *stft*() faster by running them on separate threads. But I was not able to successfully achieve my intended results here. But I was able to achieve minor improvement in this method by using Parallel.For loop for the complex array Y.

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

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

Figure 5: Sequential for loop in *stft*()

```
Parallel.For(0, wSamp / 2, new ParallelOptions { MaxDegreeOfParallelism = MainWindow.threadPool }, ll =>
{
    Y[ll] = new float[2 * (int)Math.Floor((double)N / (double)wSamp)];
});

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

Figure 6: Parallel for loop in *stft*()

4.1.2 Parallelizing *fft*()

As discussed already in the previous sections, *fft*() is a very significant method in this application, as it uses huge amount of processor resources due to its recursive implementation and also because its being called in some other methods too.

```
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;
}
```

Figure 7: Recursive implementation of *fft*()

4.1.2.1 Overcoming the barriers for fft()

In order to enhance the performance of *fft*(), firstly I tried to put the two even and odd arrays in two separate tasks. Which did not produce any fruitful results. And secondly, I tried to parallelize the for loop using Parallel.For to enable *fft*() operate faster. But that did not produce any positive outcome too as it gave an OutofRangeException error.

Then after some research along with trial and errors, I managed to redo the *fft*() algorithm in an iterative manner. In order to use an iterative *fft* algorithm, I had to replace the initial *fft*() method and replace with two new methods namely *RevereseBits*() *and Iterative*()(**APPENDIX D**). The new altered Cooley-Tukey iterative *fft* algorithm produced improvement in the run time.

Using the iterative Cooley-Tukey algorithm produced more scopes for potential parallelization. It also performs lesser computational index, compared to the recursive version, thus producing enhancement in performance.

```
1 reference
static int ReverseBits(int bit, int n)
{
    int reverse = n;
    int counter = bit - 1;

    n >>= 1;
    while (n > 0)
    {
        reverse = (reverse << 1) | (n & 1);
        counter--;
        n >>= 1;
    }
    return ((reverse << counter) & ((1 << bit) - 1));
}


2 references
public static Complex[] Iterative(Complex[] x, int L, Complex[] twiddles)
{
    int N = x.Length;
    Complex[] Y = new Complex[N];
    int bits = (int)Math.Log(N, 2);


    for (int i = 0; i < N; i++)
    {
        int pos = ReverseBits(bits, i);
        Y[i] = x[pos];
    }

    for (int ii = 2; ii <= N; ii <<= 1)
    {
        for (int jj = 0; jj < N; jj += ii)
        {
            for (int kk = 0; kk < ii / 2; kk++)
            {
                int e = jj + kk;
                int o = jj + kk + (ii / 2);
                Complex even = Y[e];
                Complex odd = Y[o];

                Y[e] = even + odd * twiddles[kk * (L / ii)];
                Y[o] = even + odd * twiddles[(kk + (ii / 2)) * (L / ii)];
            }
        }
    }
    return Y;
}
```

Figure 8: Recursive implementation of *fft*

I have also decided to remove the second implementation of the *fft*() method from the MainWindow.xaml.cs and decided to implement the new *fft* in the timefreq.cs and calling it from there whenever its required.

I also tried to parallelize the bit reversal for loop in the newly created *Iterative*() method, but parallelizing it did not improve any performance, thus implementing it in a ordinary manner.

4.2 Parallelizing MainWindow.xaml.cs

The class MainWindow.xaml.cs consumes the highest amount of processor resources, as it has its own methods and as all the other classes are being called from here. The main methods which seemed to have opportunity for parallelizing are *freqDomain*() and *onsetDetection*().

4.2.1Parallelizing *freqDomain*()

Firstly, I tried to parallelize the method *freqDomain*(), as it was consuming a huge amount of processor time to operate. Refer to **APPENDIX C** for a better understanding. After several trial and errors, I was only able to make slight improvement in this method by using Parallel.Invoke.

```
1 reference
private void freqDomain()
{
    stftRep = new timefreq(waveIn.wave, 2048);
    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];
        }
    }

}
```

Figure 9: Sequential implementation of *freqDomain()*

```
private void freqDomain()
{
    Parallel.Invoke(() =>
    {
        stftRep = new timefreq(waveIn.wave, 2048);
    });

    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];
        }
    }
}
```

Figure 10: Parallel implementation of *freqDomain()*

4.2.2 Parallelizing *onsetDetection*()

In the sequential implementation of the application, the method *onsetDetection()* was consuming a large portion of the CPU. In fact it was consuming even more CPU than the *fft()* and *stft*(), with a staggering 48.94% as seen on **Figure 2.** From the performance profiler, I noticed that some amount of CPU was being utilized for the HFC calculation block, even though it had no dependencies. However, significant amount of CPU was being used within the loop, where *fft()* from MainWindow.xaml.cs was being called,

thus ensuring this as one of the significant portion of processing and eligible for parallelization. This method also calculated *twiddles* values like the timefreq.cs class.

As for *onsetDetection*(), in the initial sequential application, there is one big loop containing 3 inner loops which are dependent on the previous one. While the second loop took *twiddles* as input the third loop's input were the complex array Y and the *fft*() was being called inside the bigger outer loop too, thus consuming more CPU.

```
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)
```

Figure 11: sequential *onsetDetecton*() loops

4.2.2.1 Overcoming the barriers for onsetDetection()

Before parallelizing anything in the *onsetDetection()*, I decided to split the 3 inner for loops into their own blocks so there could be more scope for parallelization. I required a global array of *twiddles* for both the first and second loop. The *twiddles* values were used in the second loop for creating a single array of arrays with the Y values. Another change I made was using the *nearest* values once for each loop as calculating the *nearest* values hardly consumed much time.

Before parallelizing the bigger loop though, I was able to enhance the spatial locality by using a Parallel.For instead of the sequential For loop. This implicit parallelization produced more scopes of parallelizing the new outer *ii* outer loop, which consequently produced fruitful performance results.

```
HFC = new float[stftRep.timeFreqData[0].Length];

for (int jj = 0; jj < stftRep.timeFreqData[0].Length; jj++)
{
    for (int ii = 0; ii < stftRep.wSamp / 2; ii++)
    {
        HFC[jj] = HFC[jj] + (float)Math.Pow((double)stftRep.timeFreqData[ii][jj] * ii, 2);
    }

}

float maxi = HFC.Max();
```

Figure 12: sequential HFC loop

```
HFC = new float[stftRep.timeFreqData[0].Length];

Parallel.For(0, (stftRep.timeFreqData[0].Length), (int jj, ParallelLoopState state) =>
{
    for (int ii = 0; ii < stftRep.wSamp / 2; ii++)
    {
        HFC[jj] = HFC[jj] + (float)Math.Pow((double)stftRep.timeFreqData[ii][jj] * ii, 2);
    }
});

float maxi = HFC.Max();
```

Figure 13: parallelized HFC loop

After parallelizing the HFC loop, I approached the three newly created loops for parallelizing. In the first loop I decided to calculate the array containing *twiddles* values in parallel, which were later assigned to the global variable *twiddles*, required in the second loop. So, I was able to implicitly parallelize the first loop. I used another Parallel.For implementation for the 2nd loop, which showed some improvement in the performance. I tried to parallelize the third loop too, but I was not successful in doing so. The screenshot for the source-code of these loops are seen below in the **Figure 14.** In conclusion, I was satisfied with my parallelization in this method, as in the final outcome I have seen lot of improvement in performance for the *onsetDetection*().

```
//upto here
for (int mm = 0; mm < lengths.Count; mm++)
{
    int nearest = (int)Math.Pow(2, Math.Ceiling(Math.Log(lengths[mm], 2)));
    twid = new Complex[nearest];


    Parallel.For(0, nearest, new ParallelOptions
    { MaxDegreeOfParallelism = threadPool }, pp =>
    {
        double a = 2 * pi * pp / nearest;
        twid[pp] = Complex.Pow(Complex.Exp(-i), (float)a);
    });

    twidArrays[mm] = twid;
}


Parallel.For(0, (lengths.Count), (int mm, ParallelLoopState state) =>
{
    int nearest = (int)Math.Pow(2, Math.Ceiling(Math.Log(lengths[mm], 2)));
    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;
        }
    }

    yArrays[mm] = timefreq.Iterative(compX, nearest, twidArrays[mm]);
});



for (int mm = 0; mm < lengths.Count; mm++)
{
    int nearest = (int)Math.Pow(2, Math.Ceiling(Math.Log(lengths[mm], 2)));
    absY = new double[nearest];

    double maximum = 0;
    int maxInd = 0;

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

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

Figure 14: parallelized *onsetDetecton*() loops

4.2.3 Parallelizing MainWindow Constructor

I did not find much scopes of parallelizing in the MainWindow constructor, as its main function was to mainly call other classes and methods. I tried to improve performance a bit better by implementing atomic parallelization on codes for loading the .wav file. But after using a new task to improve the performance, instead of getting enhancement, the parallelization transferred the extra overhead to the .xml file input. So, I just decided to keep this part of the MainWindow as sequential, in order to avoid the overhead. The difference can be seen on **Figure 15 & 16.**

```
 49 (0.64%)   38          InitializeComponent();
539 (7.09%)   39          filename = openFile("Select Audio (wav) file");
303 (3.98%)   40          string xmlfile = openFile("Select Score (xml) file");
              41          Thread check = new Thread(new ThreadStart(updateSlider));
 24 (0.32%)   42          loadWave(filename);
```

Figure 15: Before parallelizing the audio input.

```
 40 (0.60%)   46          InitializeComponent();
              47          Task.Factory.StartNew(() =>
              48          {
              49              filename = openFile("Select Audio (wav) file");
              50          });
666 (10.00%)  51          string xmlfile = openFile("Select Score (xml) file");
```

Figure 16: Before parallelizing the audio input.

5.0 Outcome & Comparisons

After I was satisfied with the amount of parallelization I did and was not able to parallelize anymore, I decided to take the execution time of both the sequential and the parallel implementation of the application respectively. I used few stopwatches from the pre-defined stopwatch class in C# to measure the amount of time spent, which included time starting from loading the file to displaying the interface. I also measured how much time the application spent for *onsetDetection*() and *stft*(), as they were very significant too. Screenshot of the CPU usage for the parallel application is given below:

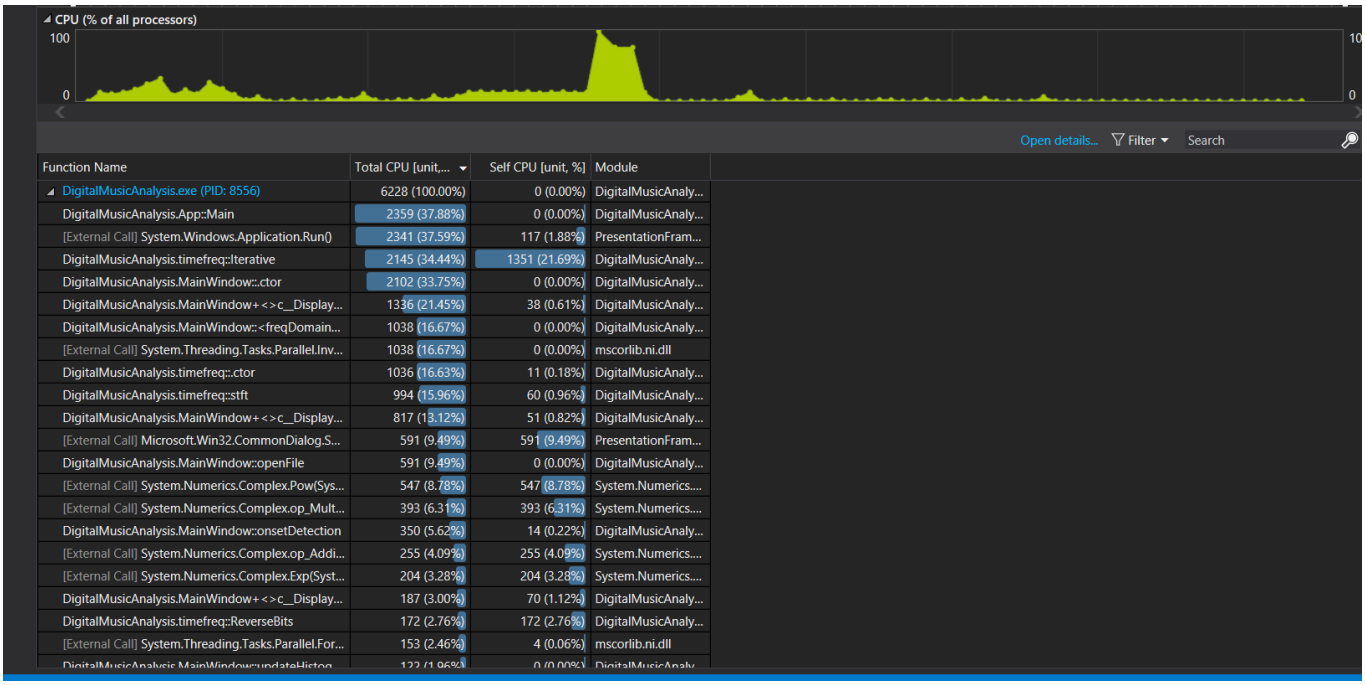

Figure 17: CPU usage for the Parallel application

In order to understand the computational speedup for the application, I first measured the time for the sequential application using stopwatch. I performed 15 tests for the sequential case, the best sequential time is recorded below:

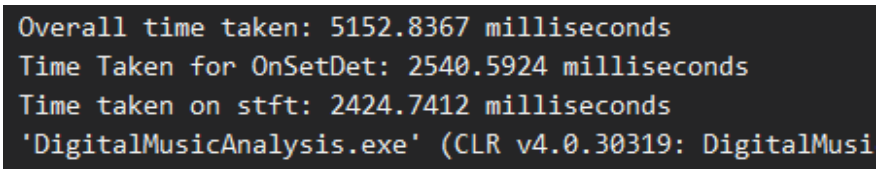

Figure 18: Time taken for best sequential case

And for the case of parallel application, I measured the time of the application by varying the number of processors up to 8. To get the best results, I performed 5 tests for each number of processors and considered only the best result. The results are given in the tables below:

Overall Runtime	
Processors	**Execution time (milli sec)**
1	2566.76

2	2185.07
3	2121.61
4	2078.6299
5	1998.5
6	1986.058
7	1897.213
8	1863.7017

Table 1: Overall runtime for each processor

Please refer to **APPENDIX E** for the execution time of *stft*() and *onsetDetection*().

5.1 Speedup

After measuring the speedup of the application using Amdahl's law, it was seen that the parallel application executed a lot faster and the speedup graph itself produced a desired result. The parallel application is 2.01x faster in a single processor and at its max limit, it is 2.76x faster than its sequential counterpart.

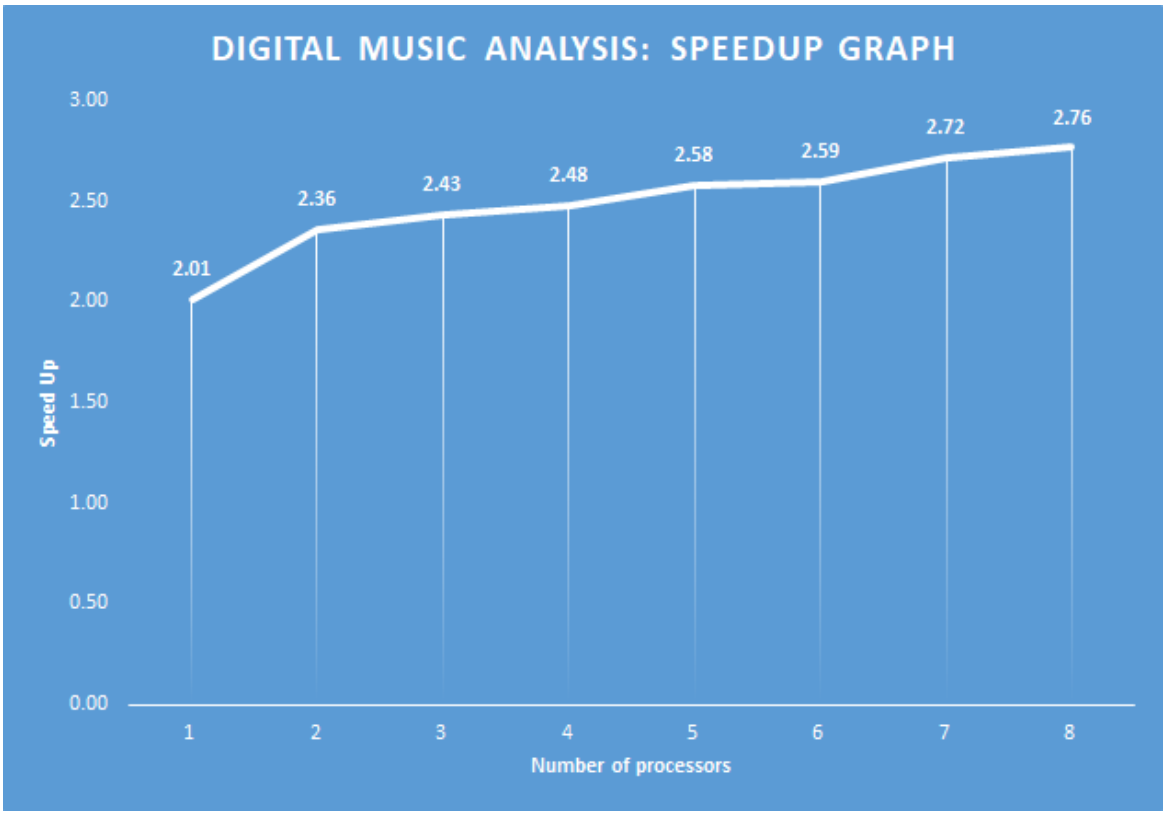

Figure 19 : Speed graph for Digital Music Analysis

The interface of the application was not interrupted due to parallelization. Please refer to **APPENDIX A** to compare the interface of the sequential and the parallel implementation. To have a better understanding of the comparison of computation for the parallel implementation, its function call graph is attached in **APPENDIX C.**

6.0 Conclusion

In conclusion, I was satisfied with the overall performance gain I was able to achieve. Though, I believe there were more scopes of potential parallelization, which I was not successful to expose. I think, there were scopes of parallelizing the *freqDomain*() method from the class MainWindow.xaml.cs and also more scopes for parallelizing *stft*().

Using the performance profiler for this project was very beneficial for me, as I believe I will be using more of it in future projects and in professional carrier. This outcomes from this parallelization project taught me about a whole new side of computing, which I was not very aware of.

7.0 Appendices

7.1 APPENDIX A

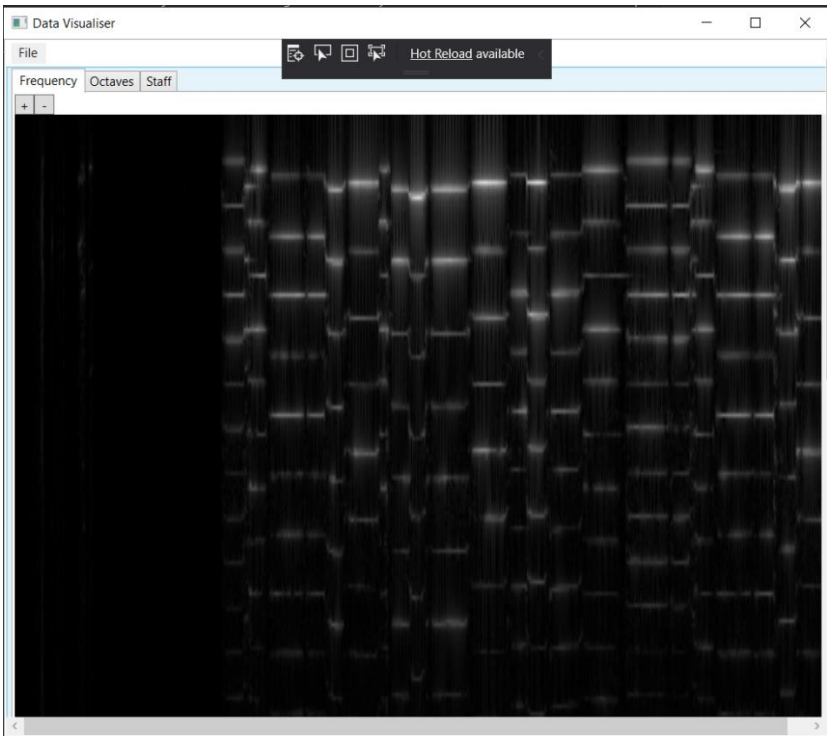

Figure: screenshot of the sequential application

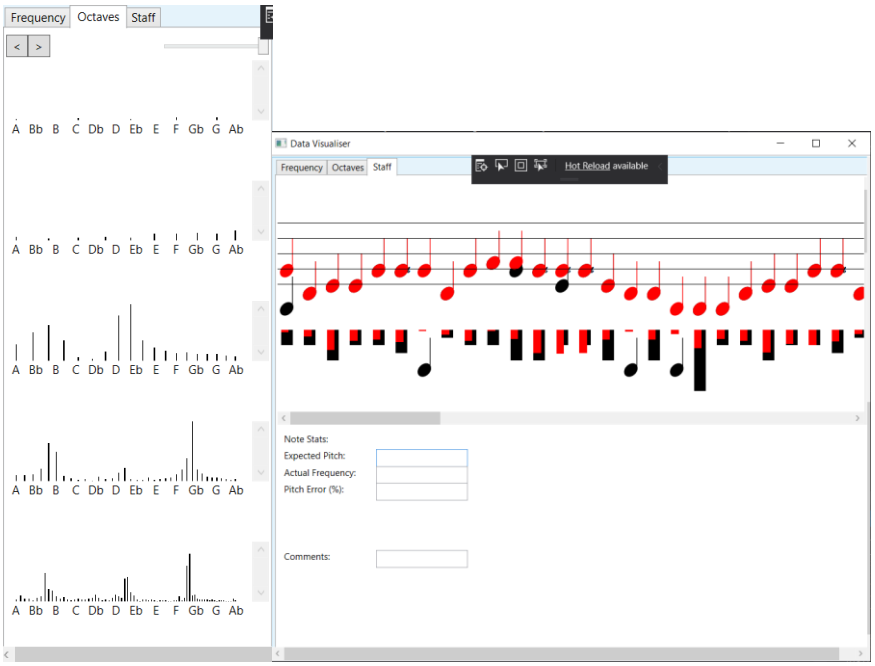

Figure: Screenshot of sequential application's Data visualiser

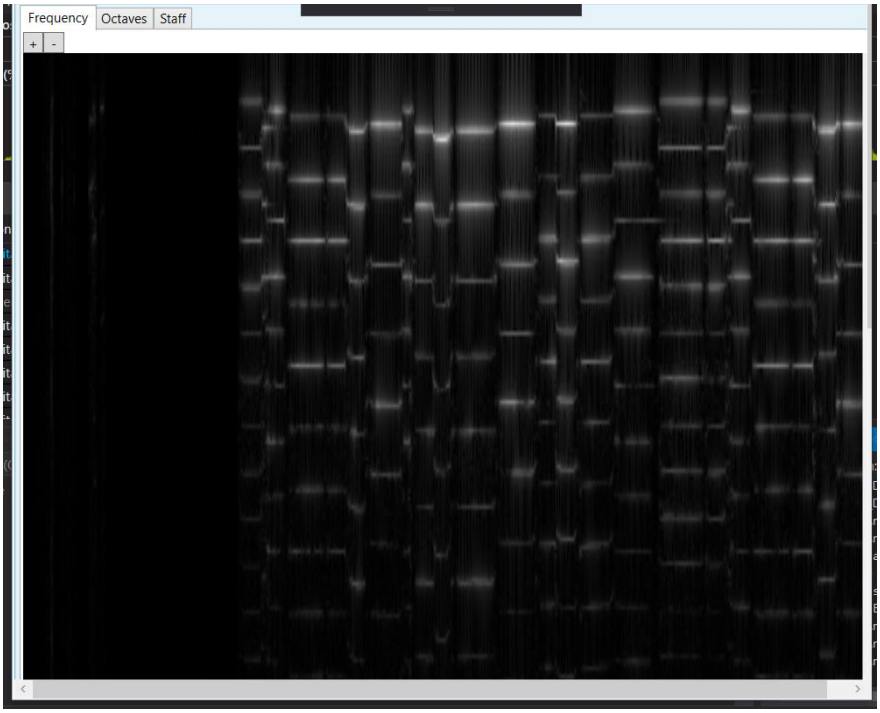

Figure: screenshot of the parallel application

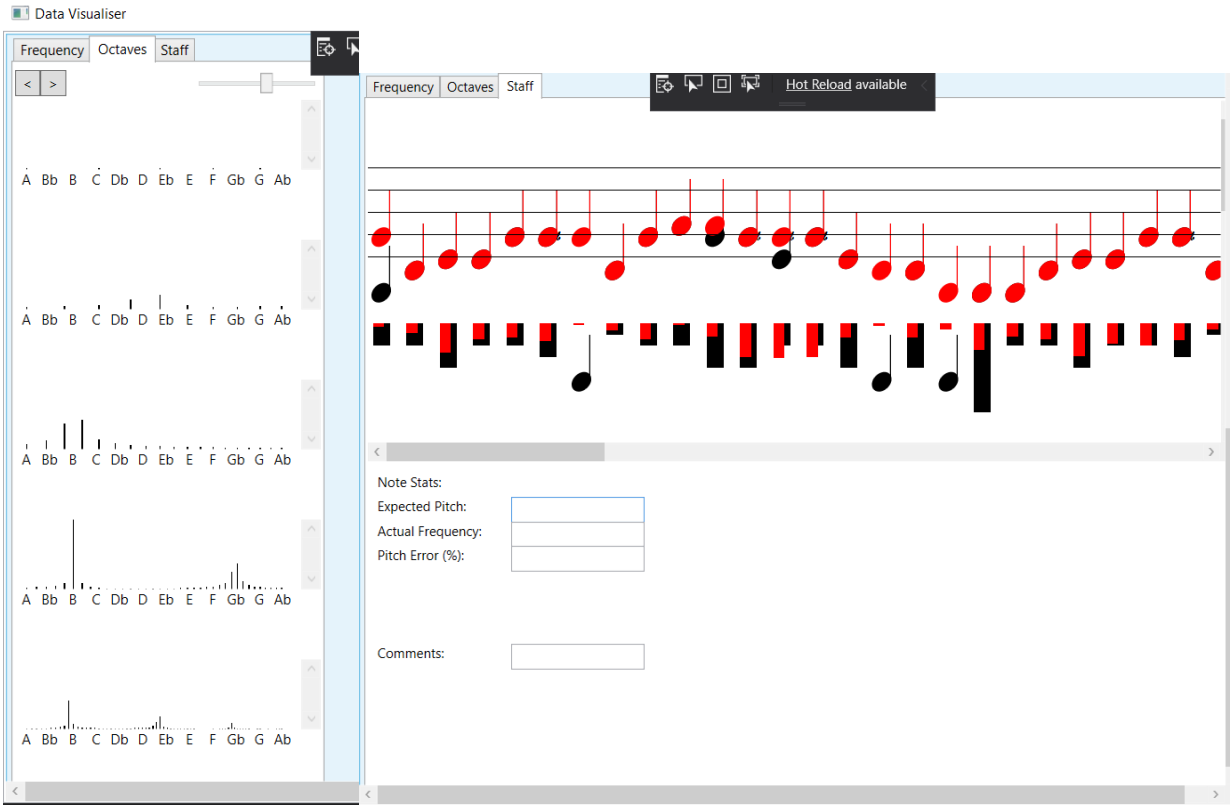

Figure: Screenshot of parallel application's Data visualiser

7.2 APPENDIX B

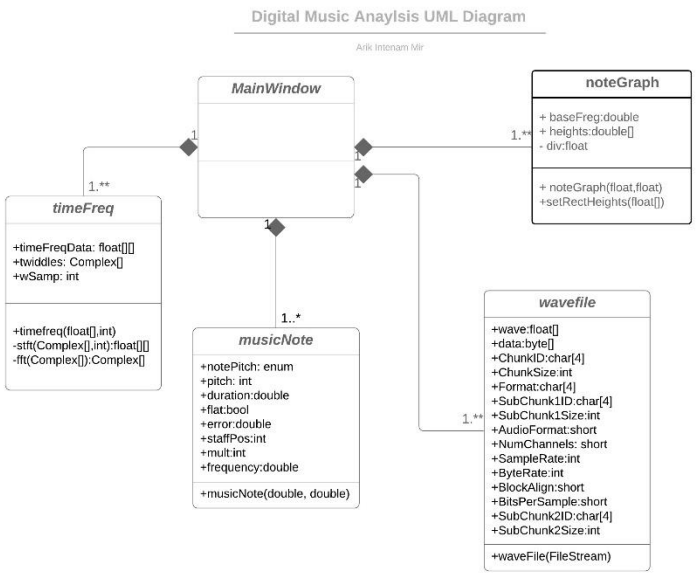

Figure: UML Diagram of the sequential application

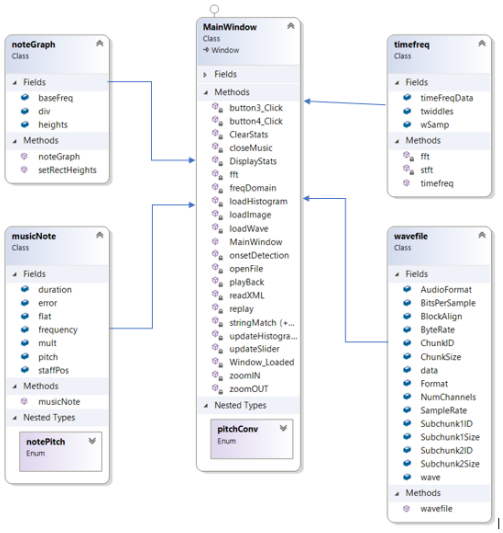

Figure: Class diagram of the sequential application

7.3 APPENDIX C

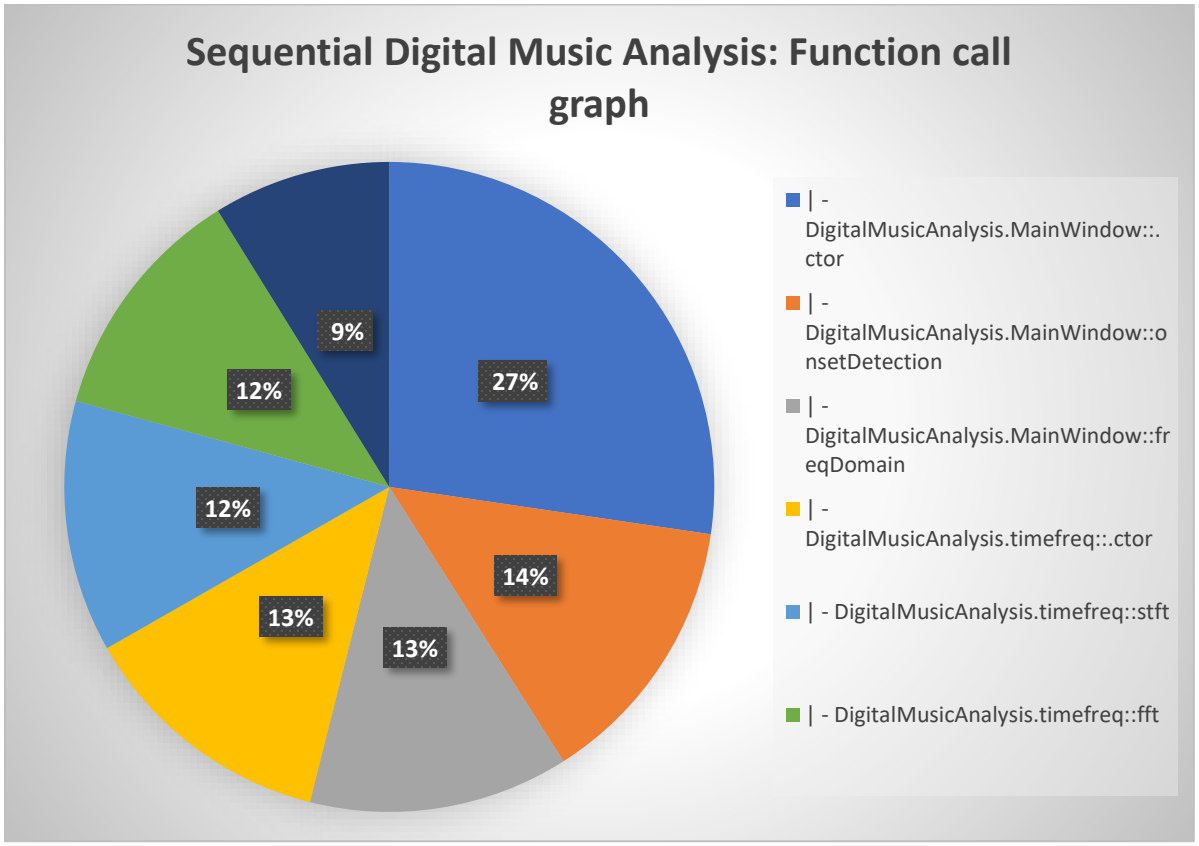

Figure: Function call graph for the sequential application

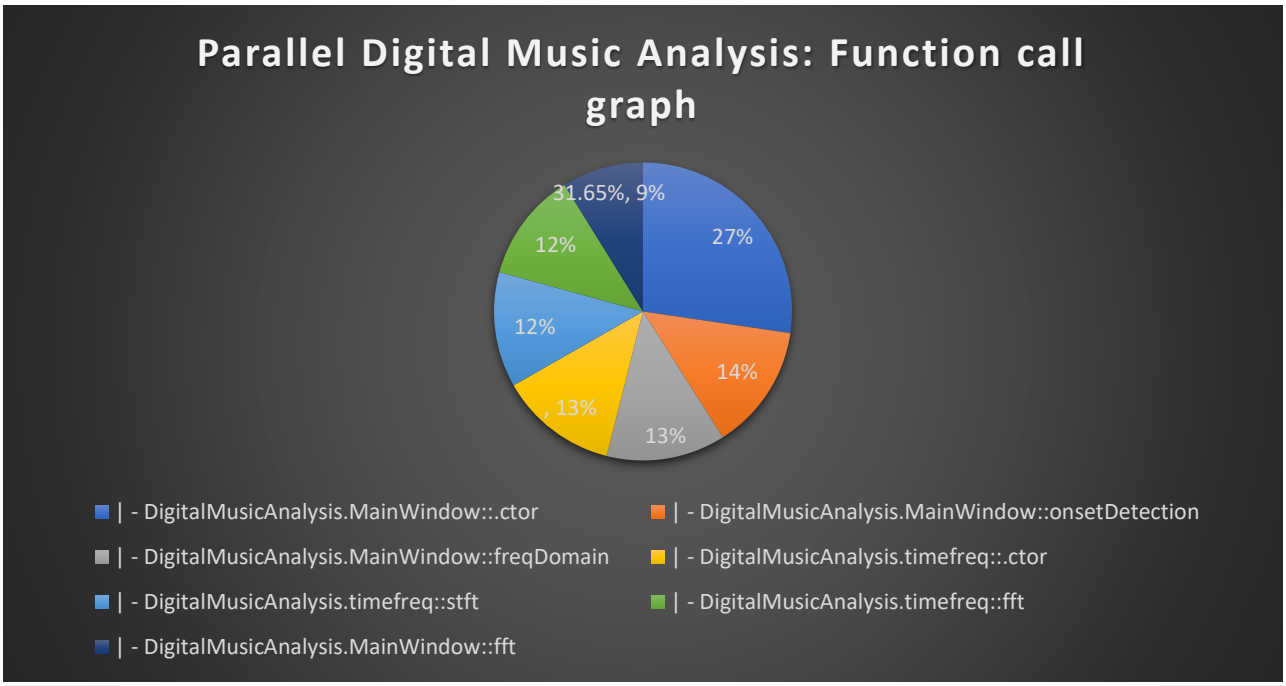

Figure: Function call graph for the parallelapplication

7.4 APPENDIX D

```
algorithm iterative-fft is
   input: Array a of n complex values where n is a power of 2
   output: Array A the DFT of a

   bit-reverse-copy(a,A)
   n ← a.length
   for s = 1 to log(n)
       m ← 2^s
       ω_m ← exp(-2πi/m)
       for k = 0 to n-1 by m
           ω ← 1
           for j = 0 to m/2 - 1
               t ← ω A[k + j + m/2]
               u ← A[k + j]
               A[k + j] ← u + t
               A[k + j + m/2] ← u - t
               ω ← ω ω_m

   return A
```

The bit-reverse-copy procedure can be implemented as follows.

```
algorithm bit-reverse-copy(a,A) is
  input: Array a of n complex values where n is a power of 2,
  output: Array A of size n

  n ← a.length
  for k = 0 to n - 1
      A[rev(k)] = a[k]
```

APPENDIX E

STFT	
Processors	**Execution time (millisec)**
1	1070.46
2	1056.38
3	1023.93
4	1116.107
5	1140.79
6	1087.6
7	1045.9

8	1033.6

Table: STFT runtime for each processor

OnsetDetection	
Num Processors	**Execution time (millisec)**
1	1130.5157
2	746.32
3	673.108
4	595.116
5	485.025
6	493.618
7	482.12
8	475.32

Table: OnsetDetection runtime for each processor