High Performance and Parallel Computing

ASSIGNMENT 1

Kishan Virani (N9307834)

Queensland University of Technology

Table of Contents

Introduction

This report presents the analysis of the Sequential and Parallel version of the program called “Digital Music Analysis”. In particular, this report will summarize the original version of the program, how it works and what it does, using the class diagram.

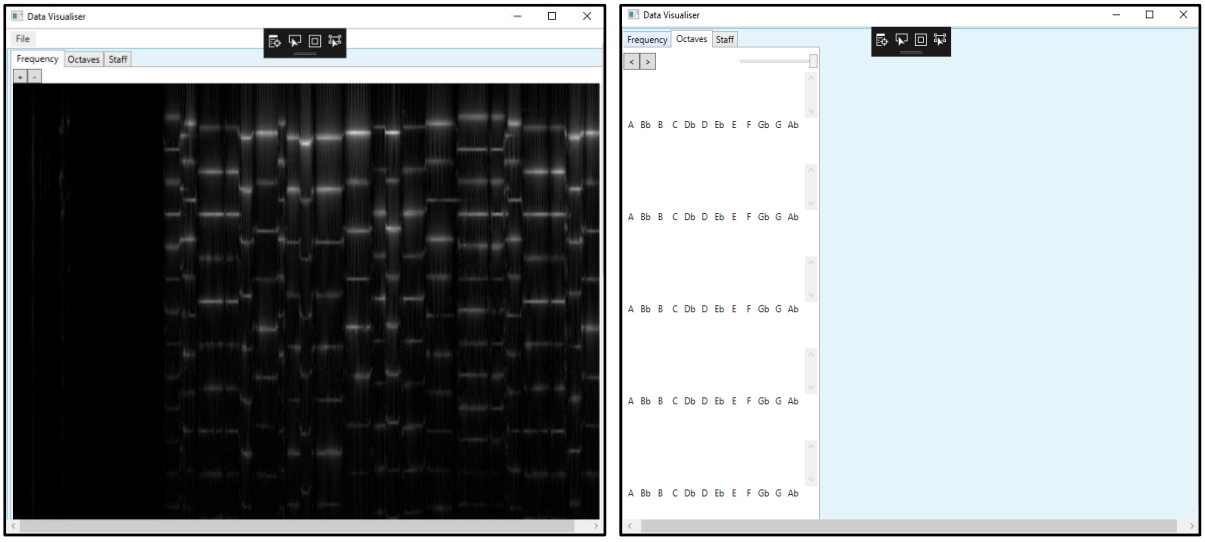
The report will also introduce parallel version of the application. It will give a detailed description of the changes, which were made to convert the sequential program into the parallel program.

Program and Testdata can be pulled using below link:

https://github.com/kishan7492/CAB401-Assignment

Introduction to Digital Music Analysis

Digital music analyser is aimed to give feedback to the music players by analysing each note played by the music learner. The main purpose behind the development of this program is to give the feedback to the students on their music in the absence of the music teacher. The program analysis what note is being played and shows the difference on to the music visualiser screen as shown below.



Analysis (Sequential Version)

What digital music analyser do?

This program takes the input of music file and sheet of music. It processes the music files and opens music visualiser where it shows the music in digital format. It shows 3 different tabs in the visualiser “Frequency”, “Octaves” and “Staff”. Frequency tab shows the frequency of the different node on the screen. Octaves tab shows that which frequency is currently playing. Staff tab shows that red and black nodes. Black node shows that the node student has played is correct, red node shows that the node student has played is incorrect. It also displays the

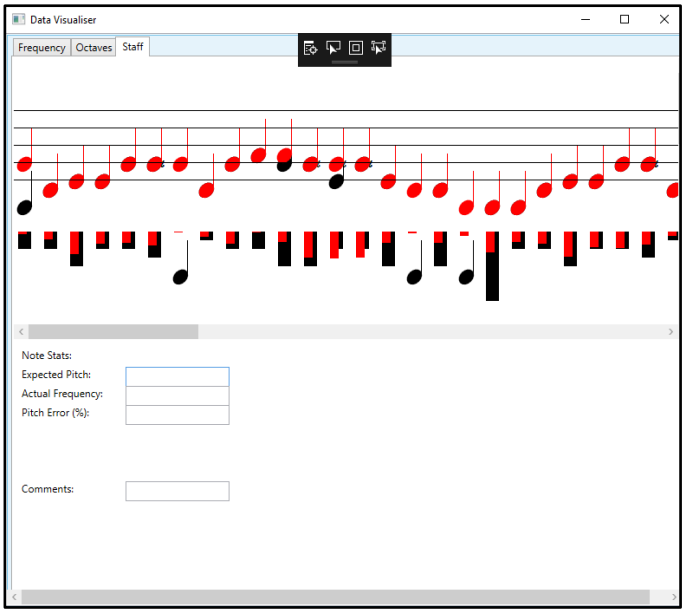


Figure 1: Digital Visualiser

pitch error in percentage whenever we hover over the nodes in the staff tab as can be seen in below picture. By looking at the pitch error, the student can make an improvement.

How Digital Music Analyser Works

The “mainWindow.xaml.cs” file contains the main function of the program. As shown in appendix A, it makes the call to the multiple functions and processes

the data. The main function of the program takes the input of the digital audio file in .wav format and sheet music file in .xml format. The program loads the digital audio file using “loadWave” function, it converts it into the big array. The main function calls the “freqDomain” function which will analyse the data that we have got from the “loadWave” function. The “freqDomain” function will call the time “timefreq” class from the timefreq.cs. “Timefreq” class transforms data into time-frequency (see appendix B for the class diagram of timefreq.cs). After converting wave file data into time-frequency array, the main function will load the sheet music file using the “readXML” function. “onsetDetection” function will analyse when the one node starts and finishes. It also shows stats in staff tab as shown in Appendix C Image 3. “loadImage” function will load the time-frequency image on tab 1 appendix C Image 1. “LoadHistogram” function loads histogram in the tab (shown in appendix C, Image 3). Visualiser also shows that which note is currently playing.

Analysis of potential parallelism

When I analysed the program, I realised that program runs one thread to update the slider. However, the program was running two major functions “freqDomain” and “onsetDetection” functions in sequential. Those two functions were doing almost all of the work of the program by detecting each note and displaying notes. By making parallelising these two functions can reduce the run time significantly.

“freqDomain” function transforms raw data into Time-frequency representation. It calls the constructor of the timefreq.cs and passed the wave file with the size of the windowSamp. First “for loop” in the constructor is running on 2048 times and places data into twiddles array Profiling data also showed that this for loop is doing heavy work. Therefore, it is worth exploiting for parallelism. According to the profiler report, there is a function call of “fft”, which does 39.70% of the work.

By examining the “fft” function I realised that it is a divide and conquer algorithm, which is designed based on the multibranch recursion. This algorithm is one of the fastest ways to find the solution in given array, therefore replacing the algorithm with other alternatives might result in the slowdown. Conclusively, I decided to parallel the “for” loop, which is calling “fft” function, instead of parallelising the “fft” function.

“onsetDetection” function measures start and finish of each note and the frequency of the notes over each duration. According to the profiler report, “Nested For” loop in line 382 of “MainWindow.xaml.cs” is doing some heavy lifting for the program. It’s filling the “HFC” array with the values of time-frequency data. So, it is safe to parallel because it’s iterating for the fixed number of times.

Another hardworking part of the “onsetDetection” is “For” loop in line 434. It is the heart of the function, it determines the start and finish of notes based on onset detection. Profiler report also showed us that it is doing almost 47% of the work. Bases on dependences analysis, there are many data dependencies are involved in this “for” loop. I believe it is still worth exploiting because it’s doing most of the work.

Data structure and Mapping

Threading allows C# program to perform synchronized processing so that we can perform multiple tasks at a time. Each thread runs independently on each processor. .NET framework provides the system.threading namespace, which makes threading task much easier. The System.Threading namespace provides multiple classes and interfaces that allow multithreaded programming.

For this program, I have created multiple threads using thread class as shown in the appendix F. I’ve used the Environment.ProcessorCount Property of the system to get the number of processors. I’ve created threads using the ProcessorCount property of the system to run the program on every core. This technique is used on both function “freqDomain” and “onsetDetection”.

The Task Parallel Library (TPL) is a set of APIs in System.Threading.Tasks namespaces. TPL allows developers to easily add parallelism in the program. TPL let the programs to use all the available processors most efficiently by handling the partitioning and scheduling of the work.

The Task Parallel Library (TPL) contains Parallel.For that are parallel versions of the normal for loop. Parallel.For can make a significant difference where dependencies do not exist between tasks and CPU is intensively used. For this

selected program, I have used the Parallel.For to split the workload of the “for” loop on every processor available.

Timing and Profiling

Before Parallelisation

Timing and Profiling

Time by functions (Time in ms) For original						
Function Name	Test1	Test2	Test3	Test4	Test5	Average time
loadwave	54	58	54	55	55	55.2
freqDomain	2618	2718	2661	2685	2604	2657.2
sheet music	3	4	3	3	9	4.4
onsetdetection	2755	2744	2679	2719	2766	2732.6
load image	5	5	6	5	5	5.2
load histogram	3	3	5	3	3	3.4
playback	79	84	92	84	89	85.6
check.start()	0	0	0	0	0	0
whole application	5522	5620	5505	5557	5507	5542.2

Table 1: Original Application running times.

To get the accurate execution times of each function and the whole program, I’ve used the Stopwatch class of the “System.Diagnostics” namespace. My objective was to understand the workload of each function via its processing timings. After recording the times of multiple tests, I calculated the average time of the whole program (shown in above table 1). The code for recording time is given in appendix E. The timing table showed that the digital music analysis is spending most of the time on functions “onsetDetection” and “freqDomain”. The average time for “freqDomain” function is 2657.2ms and the average time for the “onsetDetection” function is 2732.6ms. Finally, by parallelising only these two functions can reduce a significant amount of time. In order to understand those two functions in details, I used performance

profiler of visual studio. By analysing the performance profiler report "onsetDetection" function is spending most of the time behind the "fft" function, which is basically divide and conquer algorithm (Details are shown in appendix D).

After Parallelisation

Timing and Profiling

Time by functions (Time in ms) For parallel App for 8 logical cores						
Function Name	Test1	Test2	Test3	Test4	Test5	Average time
LOADWAVE	56	54	53	53	50	53.2
freq domain	1313	1379	1291	1258	1407	1329.6
sheet music	3	3	4	3	3	3.2
onsetdetection	2283	2271	2245	2193	2311	2260.6
load image	5	5	5	5	5	5
load histogram	5	3	3	3	3	3.4
playback	90	91	88	85	91	89
check.start()	0	0	0	0	0	0
whole program	3857	3809	3689	3603	3853	3762.2

Figure 2: Timings of function in the parallel version

Above image shows the time consumed by each function on 8 logical cores (other core timings are in testdata.xlsx in the repository on GitHub). Timing is taken as the same way it was done in sequential version. I've recorded times for the 5 tests and took the average. "freqDomain" function was taking 2657.2ms in sequential version, because of the parallelisation it is now taking the 1329.6ms. "onsetDetection" was taking 2732.6ms in sequential version, because of parallelisation it is now taking 2260.6ms.

Results

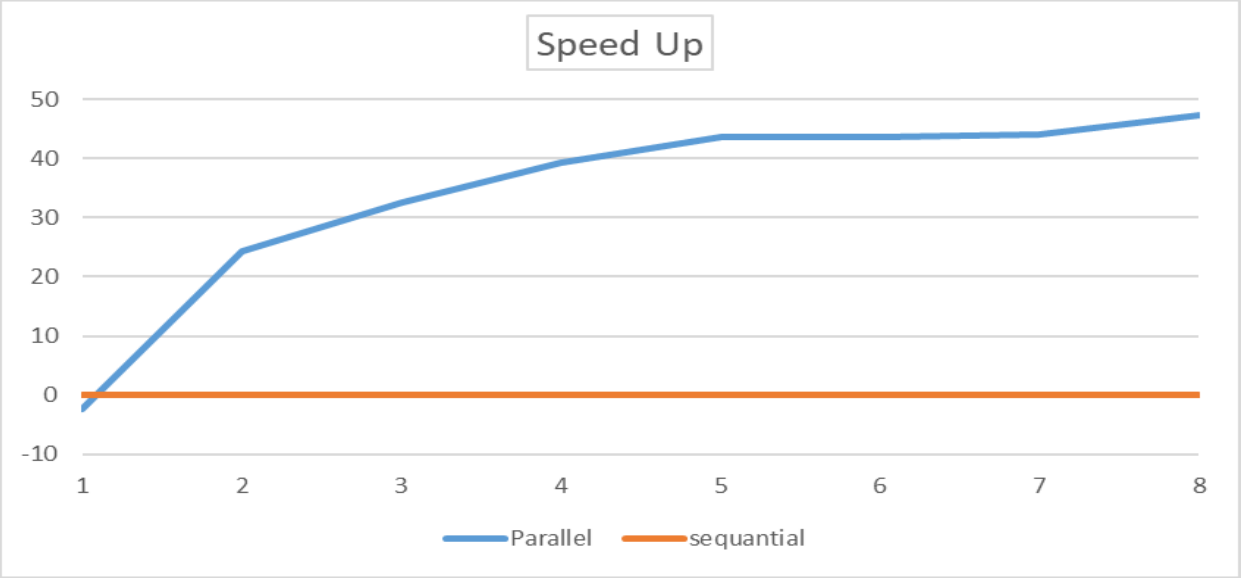


Figure 3: Speed up graph

Above graph shows speed up by the increment of the number of cores. The Outcome showed that running the application on the modern parallel computer can be beneficial as it reduces the processing time meaningfully. The speed up graph shows that the running time of application is decreasing until we reached the limit, which is 5 core. The program gave good seed up until 5 cores, after that it didn't reduce the running time. Which proves that this is sub-linear speed up of the program. I haven't tested the program on more than 8 core computer because it stopped giving improvement after 5 core. This parallelism cannot be considered as the scalable parallelism because speedup is not increasing as the number of processors increases.

Software, tools and techniques

Software

As of the program is written in C# language, I've used Microsoft's visual studio community 2017 to run the program.

Tools

CodeMap

Code Map is a component of visual studio. Which shows code and data dependencies. I used this tool to visualise data dependences of the program. This tool also helped me to understand Program's call structure.

Performance Profiler

I've used the Performance profiler of the visual studio to get the detailed report of the program's performance. Performance profiler gives a detailed report of each function's call and its system usage.

Techniques

I've used the two of the .NET classes to obtain close to the best performance of the digital music analysis.

.NET framework 4.7.2 is used to parallelise the program. It is a software framework developed by Microsoft. It includes a large class library. I've used thread class and task parallel library of .NET framework.

Development System Specification:

Processor: 6th generation Intel Core i7-6700

Code Name: SkyLake

Number of Cores: 4 Physical

Number of Logical Cores: 8 logical cores

Socket: FCLGA1151

Clock Speed: 3.40 GHz

Hyper-Threading: Yes

Memory: 16GB DDR4 3788.9MHZ

Cache: 8 MB SmartCache

Code changes in parallel version

As mentioned earlier in this report, I have made a few changes to run the program in parallel.

Changes in "onsetDetection"

Based on profiling report, for loop in line 382 of original code had noteworthy workload, therefore, I've used threading method to split workload on all the processors (As shown in appendix F). I've created threads array. Using for loop I've filled the array with the thread and started each thread. To ensure that threading does not cause any data dependency issues, I used threads.join() method to wait until all the threads finish their tasks.

Another change was made in line 438 of the original code. I changed the original for loop with Parallel.For to run for loop in parallel as shown in Appendix G.

Changes in timefreq.cs

For loop in line 19 of timefreq.cs of the original code is changed to the Parallel.For. For loop was calculating the value of wSamp and placing it into twiddles array. I used the Parallel.For because there was no data dependency and it was using a noticeable amount of CPU (As shown in Appendix H).

The workload of for loop in line 74 of timefreq.cs of the original code was reduced using the threading. The Same approach of threading is used here as mentioned above in “onsetDetection” method (As shown in Appendix I).

Overcoming the barriers

After finalising the program to parallel, the first complication that I faced was to understand the logic of the entire program. It took me a while to understand the program, but I was not entirely sure how the program is processing the music file. After struggling for a few weeks, I saw that Wayne uploaded the video explaining the application in details, it helped me a lot to understand the logic behind the program.

The Second obstacle came on my way when I chose to make a separate class of fft function instead of repeating same fft function in both classes. In both classes, the biggest issue was data dependency. Due to multiple data dependences, I dropped the thought of making another class of fft function.

The final and biggest stumbling block was my personal computer, which had only 4 logical cores. The assignment requirements were to test parallel application on more than 4 cores. Therefore, I had to come to the university to work on the assignment.

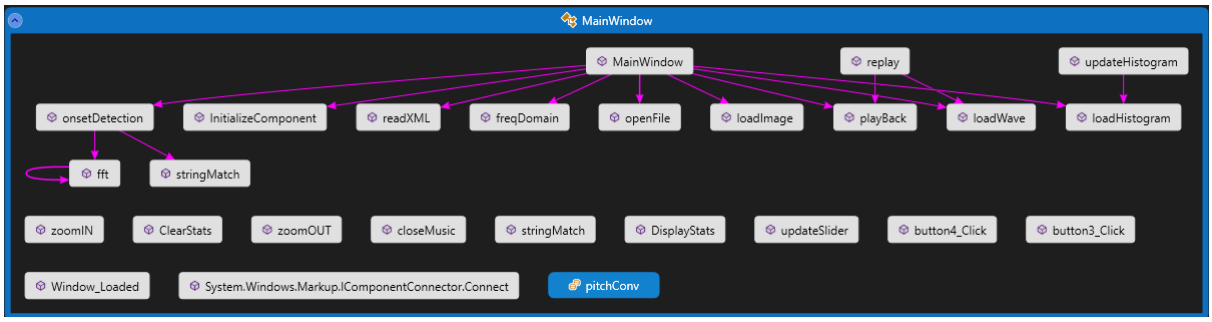
Reflection

Before starting this subject, I had some idea about the word “parallel” in context of computers. Therefore I chose this unit to get better understanding of parallelism. This unit provided me exceptional knowledge regarding various frameworks and performance improvement libraries.

The attempt to parallelise digital music analyser was accomplished positively, where I managed to get a sub-linear speedup. I was able to get almost 47% of the reduction in application runtime.

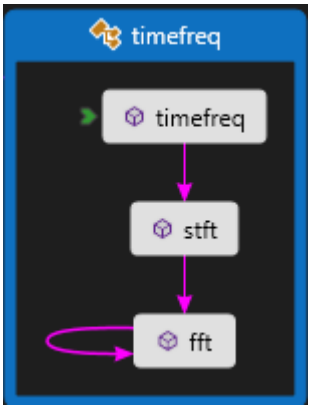
I believe changing the algorithm of the fft function could have improved the efficiency of the program. If I had more time, I could have tried to rewrite the algorithm behind the fft function.

Appendix A: MainWindow.xaml.cs class diagram



Class diagram of mainWindow.xaml.cs. The pink arrow shows the call made by mainWindow to the other function.

Appendix B: timefreq.cs Class Diagram



Class diagram of Timefreq.cs. This class contains 2 functions and one constructor.

Appendix C: Screens of the data visualiser

Image 1: Shows time-frequency of the music.

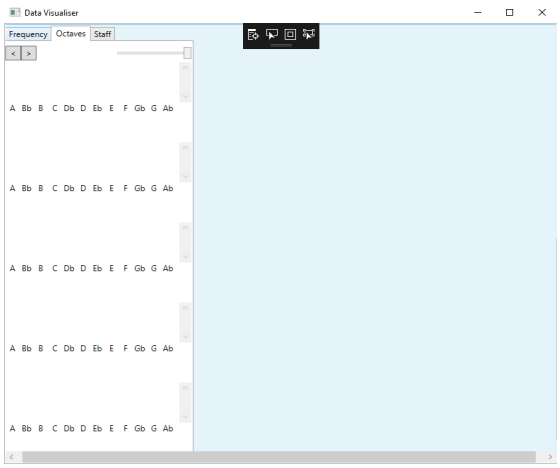


Image 2: Shows the details of each currently playing octaves

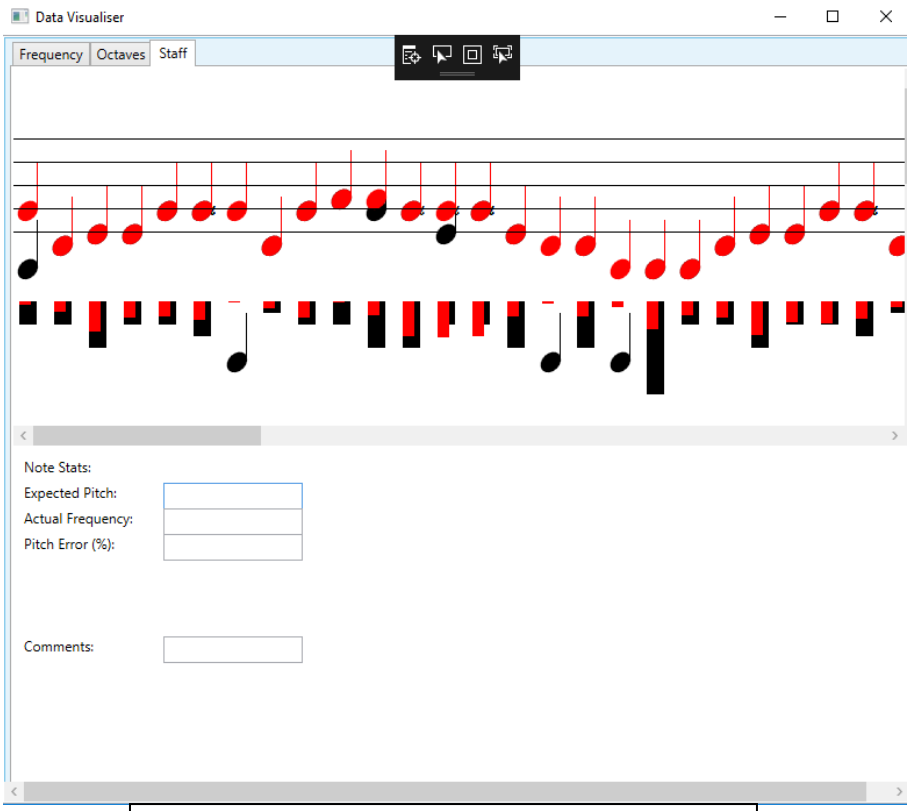


Image 3: Shows stats of each notes. Which includes pitch error in percentage. With the tip in comments

Appendix D: Profiler report of “onsetDetection”

Above image is showing the profiler report of “onsetDetection”, which shows that “fft” function in mainWindow.cs is consuming most of the time.

Appendix E: Timing the functions

```
//starting timer for whole program
var timerwholeprogram = new Stopwatch();
timerwholeprogram.Start();
Console.Out.Write("timer started for whole progeram\n");

//starting timer for loadwave function
var timerLoadWave = new Stopwatch();
timerLoadWave.Start();
Console.Out.Write("timer started for load wave \n");
loadWave(filename);
timerLoadWave.Stop();
Console.Out.Write("loadwave timer ended. Time elapsed: {0} \n",timerLoadWave.ElapsedMilliseconds);

//starting timer for freqdomain function
var timerfreqdomain = new Stopwatch();
timerfreqdomain.Start();
Console.Out.Write("timer started for freqdomain \n");
freqDomain();
timerfreqdomain.Stop();
Console.Out.Write("freqdomain timer ended. Time elapsed: {0} \n", timerfreqdomain.ElapsedMilliseconds);

//starting timer for sheetmusic function
var timersheetmusic = new Stopwatch();
timersheetmusic.Start();
Console.Out.Write("timer started for sheet music \n");
sheetmusic = readXML(xmlfile);
timersheetmusic.Stop();
Console.Out.Write("sheetmusic  timer ended. Time elapsed: {0} \n", timersheetmusic.ElapsedMilliseconds);



//starting timer for onsetdetection function
var timeronsetdetection = new Stopwatch();
timeronsetdetection.Start();
Console.Out.Write("timer started for onsetdetection  \n");
onsetDetection();
timeronsetdetection.Stop();
Console.Out.Write("onsetdetection  timer ended. Time elapsed: {0} \n", timeronsetdetection.ElapsedMilliseconds);


//starting timer for loadimage function
var timerloadimage = new Stopwatch();
timerloadimage.Start();
Console.Out.Write("timer started for load image  \n");
loadImage();
timerloadimage.Stop();
Console.Out.Write("loadimage  timer ended. Time elapsed: {0} \n", timerloadimage.ElapsedMilliseconds);
```

Time for both programs is recorded using the stopwatch class as shown in the image.

Appendix F: For loop of onsetDetection function

```
////////////////////////////////////////////////////////////////////////////
// Createing an array of thread with the size of number of Processor
Thread[] threads = new Thread[numberofprocessor];

for (int thread = 0; thread < numberofprocessor; thread++)
{

    //creating new thread that calls a parameterized method doworkonsetforloop.
    threads[thread] = new Thread(DoWorkOnSetForLoop);
    threads[thread].Start(thread);
}
// Join all the threads.
for (int thread = 0; thread < numberofprocessor; thread++)
{
    threads[thread].Join();
    //System.Console.Out.Write("thread  joined {0}\n", thread);
}

////////////////////////////////////////////////////////////////////////////
timeronsetfor.Stop();
Console.Out.Write("onset for loop  timer ended. Time elapsed: {0} \n", timeronsetfor.ElapsedMilliseconds);
```

```
1 reference
public void DoWorkOnSetForLoop(object data)
{
    int threadId = (int)data;
    int CHUNK_SIZE = (stftRep.timeFreqData[0].Length) / numberofprocessor;
    int start = threadId * CHUNK_SIZE;
    int finish = Math.Min(start + CHUNK_SIZE, stftRep.timeFreqData[0].Length);
    //System.Console.Out.Write("doworkonsetforloop thread {0}, {1} -> {2}\n", threadId, start, finish);
    for (int jj = start; jj < finish; jj++)
    {
        for (int ii = 0; ii < stftRep.wSamp / 2; ii++)
        {
            HFC[jj] = HFC[jj] + (float)Math.Pow((double)stftRep.timeFreqData[ii][jj] * ii, 2);
        }

    }
}
```

Appendix G: Parallel.For loop of onsetDetection

```
        twiddles = new Complex[nearest];

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

        compX = new Complex[nearest];
```

Appendix H: Parallel.For loop of timefreq constructor

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

Appendix I: Parallel.For loop of timefreq constructor

```
var timertimefreqfftcalling = new Stopwatch();
timertimefreqfftcalling.Start();
Console.Out.Write("timer started for timefreq fftcalling part  \n");

//////////////////////////////////////////////////////////////////////////
///


// Createing an array of thread with the size of number of Processor
Thread[] threadsarray = new Thread[numberofprocessor];

for (int thread = 0; thread < numberofprocessor; thread++)
{

    //creating new thread that calls a parameterized method fftcallingforstft.
    threadsarray[thread] = new Thread(fftcallingforstft);
    threadsarray[thread].Start(thread);
}
// Join all the threads.
for (int thread = 0; thread < numberofprocessor; thread++)
{
    threadsarray[thread].Join();
    //System.Console.Out.Write("thread  joined {0}\n", thread);
}


//////////////////////////////////////////////////////////////////////////
///
timertimefreqfftcalling.Stop();
Console.Out.Write("timefreq fftcalling part  timer ended. Time elapsed: {0} \n", timertimefreqfftcalling.ElapsedMilliseconds);
```

```
1 reference
public void fftcallingforstft(object data)
{

    int threadId = (int)data;
    int CHUNK_SIZE = (2 * (int)Math.Floor(N / (double)wSamp) - 1) / numberofprocessor;
    int start = threadId * CHUNK_SIZE;
    int finish = Math.Min(start + CHUNK_SIZE, (2 * (int)Math.Floor(N / (double)wSamp) - 1));
   // System.Console.Out.Write("fftcalling for stft thread {0}, {1} -> {2}\n", threadId, start, finish);
    Complex[] temp = new Complex[wSamp];
    Complex[] tempFFT = new Complex[wSamp];

    for (int ii = start; ii < finish; ii++)
    {

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


    }


}
```

Appendix J: Entire Program Structure