

Digital Music Parallelisation

INB375 Parallel Computing



September 27, 2015

Damon Jones n8857954

Chris Litherland N8849269

<https://github.com/dittopower/INB375-Parallel-Computing>

Contents

[Digital Music 2](#_Toc433923213)

[Black box 2](#_Toc433923214)

[How it works 2](#_Toc433923215)

[Potential Parallelism Analysis 3](#_Toc433923216)

[freqDomain() 3](#_Toc433923217)

[onsetDetection() 3](#_Toc433923218)

[fft() 3](#_Toc433923219)

[Parallelisation Process 4](#_Toc433923220)

[Results & Speedup 5](#_Toc433923221)

[Results 5](#_Toc433923222)

[Graph 5](#_Toc433923223)

[Tools & Techniques 6](#_Toc433923224)

[Tools 6](#_Toc433923225)

[Speedup Confirmation Procedure 6](#_Toc433923226)

[Hardware 6](#_Toc433923227)

[Issues & Solutions 7](#_Toc433923228)

[Granularity 7](#_Toc433923229)

[Data Dependency 7](#_Toc433923230)

[Code Explanation 8](#_Toc433923231)

[fft() 8](#_Toc433923232)

[Timefreq.cs 8](#_Toc433923233)

[Mainwindow.xaml.cs 9](#_Toc433923236)

[Reflection 11](#_Toc433923239)

[Appendix 12](#_Toc433923240)

[Digital Music GUI 12](#_Toc433923241)

[Class diagram 13](#_Toc433923242)

[Visual Studio Diagnostic Session 14](#_Toc433923243)

[Sequential Timings 15](#_Toc433923244)

# Digital Music

## Black box

The Music analyser program takes a music file in wav format and analyses the contents to find and display the frequency spectrum, notes & pitch of each note. First the user selects a music recording in WAV format and then the XML score representation of the same piece. The program then compares the recording to the score to determine the player’s accuracy. Once analysed, the GUI appears showing the octave and note distribution for each sample point. The GUI also shows a graphical staff representation of the notes played overlayed on the actual notes in the score. See Appendix: Digital Music GUI for screenshots of the Interface.

## How it works

Digital Music Analyser compares the frequency and pitch of the sheet music XML to a supplied audio recording of the player.

The Main method loads the wave file as a stream and creates a new ‘wavefile’ object. The **wavefile** class converts the file into a float array of the binary bits.

The next method call is freqDomain() which transforms the wave file into a time-frequency representation as pixelArray[]. This initialises the stftRep variable as a new instance of the timeFreq class. The timeFreq class is responsible for processing the raw music file data into a time frequency array. This is accomplished by using the fft() function to find the pitch of the note and assign it to the current sample point within the specified range (default is 2048).

Once this wave is loaded, the XML sheet is analysed next. ReadXML() creates an array of **musicNote** objects, similar to above. However, it also creates several *Lists* as well for tracking octaves and duration counts. The **musicNote** object has some public parameters such as *notePitch*, *duration* and *frequency*.

Having completed initialisation, the last method onSetDetection() is called to perform the analysis of the imported wave file against the XML.

Once analysis in onsetDetection is complete, the GUI is constructed using a variety of functions to construct and display the various Windows Forms representations of the analysed data within the MainWindow.

# Potential Parallelism Analysis

Timing code was added to the program and used to time the various sections of the main method. These timings create an idea of the relative execution times of different core functions and identify which of these require significant execution time and could benefit most from parallelisation. See [Appendix: Sequential Timings](#_Sequential_Timings).

## freqDomain()

This is one of the two most time consuming functions in the program, it is composed of a nested loop and a call to the time frequency class where the most of the execution time occurs. This loop and considerable execution time suggests a potential for significant performance gains when parallelised.

Inside timefreq() the main problem is a large loop which calls the recursive function fft(), See fft(). Rather than parallelising fft(), the alternative would be to paralyse the loop which calls fft(). This parallelisation method would not be as effective but would still provide an easily achievable significant boost. This method would also scale well, potentially providing performance gains when executed in parallel across the iteration space of the loop.

## onsetDetection()

The time analysis for onsetDetection highlights that the 5th loop (‘mm’) accounts for almost the entirety of the function runtime. The main time consumer of this loop is a further call to the oft-referenced fft() function, See [fft()](#_fft__). Again, the potential for parallelisation can either be within fft() or for the containing mm loop. The only concern about parallelising the mm loop is the final output to the *pitches* list which must be kept in order. There are other global variables depended on within mm, however these can be converted to local variables for each iteration or are read only and no potential for mutation during the loop execution exists.

## fft()

The fft() function is one of the most complex structures in the application when it comes to parallelisation. The function itself is called within onsetDetection and freqDomain. The function itself also contains multiple loops and makes recursive calls to itself twice. This results in a significant opportunity for parallelisation as most of the procedures within fft() are repetitive. However, the recursive nature of the loop means that fft() would require a significant refactoring in order to be efficiently parallelisable. The major issue with this function is that it is called many times by the surrounding loops in addition to its recursive nature, leading to the large overall time consumption. It is unlikely that refactoring this function would deliver a worthwhile benefit compared with the other parallelisation opportunities and the required effort to parallelise fft safely.

# Parallelisation Process

The main process used to achieve parallelisation is .NET Threads. The C# runtime and Windows Operating System handle the physical assignment of these threads to each available processing node. This is predicated by setting the max\_threads constant in the program to the number of cores available to the system. For most modern quad core processors, a 4 thread setting here will achieve an optimal granularity with minimal overhead. In our testing, we initially attempted to map the threads to the 8 virtual cores available to our testing systems. However, upon trying different combinations, we found that setting to the physical machine cores yields the best results.

The data from the parallelised loop is divided into roughly even blocks with one block being given to each thread to balance the workload efficiently. The general formula we’re using to spilt the work is Blocksize = (Maximum Loop Size + No. of threads - 1) / No. of threads.

The wave data processed in the ‘mm’ loop within onsetDetection must be ordered chronologically. As this loop is parallelised, the output can return in any order from the threads and needs to be synchronised. Each thread is responsible for a subset block of the data assigned to that iteration. Synchronisation is accomplished using the .NET Threads join function which blocks the main thread until the all threads are completed for this loop. Once complete, each sub-list is joined back into the master list and is in the correct order.

# Results & Speedup

To compare the final version with the sequential program, binary files were generated and archived for later performance testing. For these final results, the original sequential binary is timed and then compared to the final parallel binary timings outputted to the console.

Majority of diagnostic timing was performed on an older ‘Bulldozer’ architecture AMD processor (FX8120). The parallel program was configured to generate 8 .NET Threads, matching the 8 cores available to the system. Timing was also performed on an additional 2 Intel based systems, a quad core desktop i5 and laptop Pentium.

## Results

### Best results as achieved on the 8 core Bulldozer architecture

Minimum sequential program: 10845 ms ~ 10714959906 ns

Minimum parallel program: 4321 ms ~ 4269367893 ns

Speedup = 10714959906 / 4269367893 = 2.5097 = 250.97% improvement

The parallelised version executes in almost 1/3 of the time of the sequential program.

Given the results obtained using a basic parallelisation technique for .NET Threads without using more advanced techniques like thread pools, it is obvious that this application is embarrassingly parallel.

## Graph

\*Maximum is the theoretical maximum based on Amdahl’s Law (1/(1-P)+(P/Processors)) where conservatively P ~= 90% of the program being parallelisable.

# Tools & Techniques

## Tools

Visual Studio is the compiler and IDE used as the project is written in C# and offers numerous tools to help with code analysis. These tools include the Diagnostic sessions which record CPU usages through the program, Code maps which help illustrate data and function dependencies between methods and classes.

Text editing programs Notepad++ and Microsoft Word were also used to help identify data dependencies in loops using their ability to highlight variables and identify references.

## Speedup Confirmation Procedure

To ensure each refactoring is performed correctly and resulted in the expected speedup, the following test procedure was used:

1. Time the existing code, find longer duration sections.
2. Duplicate the existing executable
3. Refactor code for parallel execution
4. Save as a new executable
5. Run the original and new executable multiple times each and compare mean results
6. If speedup is identified, commit new version, otherwise return to step 3.

## Hardware Specification for results testing

|  |  |  |  |
| --- | --- | --- | --- |
| System | System 1 | System 2 | System 3 |
| Type | Desktop | Desktop | Laptop |
| CPU Architecture | Bulldozer | Sandy Bridge | Silvermont |
| CPU Brand | AMD | Intel | Intel |
| CPU Model | FX 8120 | 2500K | N3530 |
| CPU Physical Cores | 8 | 4 | 4 |
| CPU Logical Cores | 8 | 8 | 4 |
| CPU Frequency | 3100Mhz | 3.3GHz | 2.16Ghz |
| Hyper threaded? | No | No | no |
| RAM type | DDR3 | DDR3 | DDR3 |
| RAM Quantity | 32gb | 16gb | 4gb |
| RAM Speed | 1333MHz | 1333MHz | -- |

# Issues & Solutions

## Granularity

Determining the most efficient thread count to use delivered unusual results. We anticipated that the most efficient arrangement would be to map each .NET thread to a virtual core. However, when testing, we found that increasing the thread count beyond 4 threads resulted in higher run times and worse speedup results. In contrast, when tested on an 8 core machine, 8 threads ran most efficiently. This raised an issue with the program running inefficiently unless the source is adjusted and recompiled. This is obviously not feasible for a user to do, so a future revision reads the system environment variable which contains the number of logical cores available for max\_threads.

## Data Dependency

One problem we encountered was the data dependency of the mm loop (the 5th loop) in the onSetDetection function. Most variable dependencies could be solved by making the variable local to each thread or elevating the variable to a class variable. However, in this case the final output of the mm loop was a series of writes to List<double> pitches. Pitches is intended to be a chronologically ordered list of the pitches or notes as they occur throughout a given audio file, so it is very important that they remain in order. The issue arose when threads would write to pitches in an unpredictable order, resulting in the list being incorrectly sequenced.

A control method was necessary to prevent this issue. One potential method was applying a lock to force threads to write to pitches in chronological order. This affected the parallelisation of this section of code and invalidated any performance benefits. Instead each thread was given its own list to output to in the correct order as it iterated through the dataset and these still-ordered sub-lists could then be combined with the other threads when each thread within the loop was complete.

# Code Explanation

## fft()

As we were looking at refactoring the fft function and there were identical versions in different locations throughout the program, fft was moved to its own class ‘Core.cs’ to reduce redundancy.

The Complex[] twiddles variable is replaced by the initialisation of the class Core core = new Core(). The setting of the twiddles variable is then “core.twiddles =” instead of “twiddles =” and calling fft is then core.fft() instead of fft().

## Timefreq.cs

### timefreq()

The variables Complex[] compX, float[] x & int nearest were moved from local variables in the constructor to class variables. This allows the parallelisation of a minor loop which sets up this variable for the operation of this class. The loop that iterates up to the nearest variable assigns the values of compX based on the corresponding xx value. This was implemented as a basic thread as shown in Figure 1*Figure 1: Basic Thread Use*Source lines are shown on 34-44 (~10 lines), Parallel lines outlined from 45-54 115-134 (~28 lines).

**Average execution times**  
*Sequential* 39286576 ns ~39ms  
*Threaded* 20370042 ns ~20ms.  
*Approximate speedup* 1.93 ns ~20ms. Approximate speed increase 1.93

Figure 1: Basic Thread Use

Thread[] mine = new Thread[MainWindow.Num\_threads];

for (int a = 0; a < MainWindow.Num\_threads; a++)

{

mine[a] = new Thread(loop);

mine[a].Start(a);

}

for (int a = 0; a < MainWindow.Num\_threads; a++)

{

mine[a].Join();

}

private void loop(object tid)

{

int id = (int)tid;

//where nearest is the maximum bound of the source loop

int blocksize = (nearest + MainWindow.Num\_threads - 1) / MainWindow.Num\_threads;

int lowerbound = id \* blocksize;

//where nearest is the maximum bound of the source loop

int upperbound = Math.Min(lowerbound + blocksize, nearest);

for (int kk = lowerbound; kk < upperbound; kk++)

{

//Code from inside the source loop

}

}

### Stft()

The variables, float fftMax, int N, float[][] Y & Complex[] x were moved from local variables within stft to class variables so that the thread could access them safely. The variables Complex[] temp, Complex[] tempFFT along with the content of a major loop were moved into a basic thread. See [**Figure 1: Basic Thread Use**](#_Figure:_Basic_Thread).

See Figure: Basic Thread Use. Source lines 74-98 (~24 lines), Parallel lines 87-95 153-182 (~37 lines).

**Average execution time**  
*Sequential* *6580505505 ns ~6581ms*   
*Threaded* *2053853011 ns ~2054ms*.  
*Approximate speedup* 3.20

The final loop of stft which reduces the results of the previous loop by the largest value was moved to a basic thread, no variable relocation is necessary as the variables were already adjusted for use in the previous parallelisation attempt. See [**Figure 1: Basic Thread Use**](#_Figure:_Basic_Thread).

See Figure: Basic Thread Use. Source lines 100-106 (~6 lines), Parallel lines 101-109 136-151 (~23 lines)

**Average execution time**Sequential *72126712 ns ~72ms*   
Threaded *26880882 ns ~27ms*.  
Approximate speedup 2.69

## Mainwindow.xaml.cs

In the main class of the program, the constant variable public const int Num\_threads = 8 was added to control the number of threads used by each parallel section of the program.

### playBack()

In the playback method there was an unnecessary variable & loop duplicated existing data to another variable of the same type without reason. To improve code, this loop was removed. This was not a parallelisation attempt and this refactoring was not used for comparison timings. Removed Source lines 698-704 (~6 lines), sound replaced with waveIn.data Source line 708.

### onSetDetection()

Local variable float[] HFC was moved to a class variable for thread access. The first loop of onSetDetection was moved to a basic thread. See Figure: Basic Thread Use.

Source lines 340-347 (~7 lines), Parallel lines 355-364 613-628 (~24 lines)

**Average execution time**Sequential *279436591 ns ~279ms*   
Threaded *69806380 ns ~70ms*  
Approximate speed increase 4.00

The local variables List<int> lengths, double pi = 3.14159265, Complex i = Complex.ImaginaryOne, List<int> noteStarts & List<double>[] allpitches were made into class variables so they would be thread readable. It was considered to swap the hard coded pi for the System.Math.PI as it would be more accurate, but was not done as we assume the original programmer must have had a reason to use a shorter version of pi. The mm loop in onsetDetection was turned into a basic thread with a couple of differences. The local variables Complex[] Y, double absY & Complex[] compX were localised to each thread to prevent dependency conflicts. The output variable was also changed from the pitches list to an array of small lists (One list for each thread). After each thread was joined back into the main thread its list was added from the array of lists back into the master list pitches.

for (int a = 0; a < Num\_threads; a++)

{

mine[a].Join();

pitches.AddRange(allpitches[a]);

}

See [**Figure 1: Basic Thread Use**](#_Figure:_Basic_Thread).  
Source lines 392-463 (~71 lines), Parallel lines 404-420 630-719 (~105 lines)  
Average execution time  
Sequential *5601776684 ns ~5602ms*Threaded *2577045212 ns ~2577ms*  
Approximate speed increase 2.17

# Reflection

During the development of the project, we discovered the difficulty in parallelising recursive functions. A particular example of this is fft() which calls itself multiple times, making any parallelisation essentially impossible with the current state of the method.

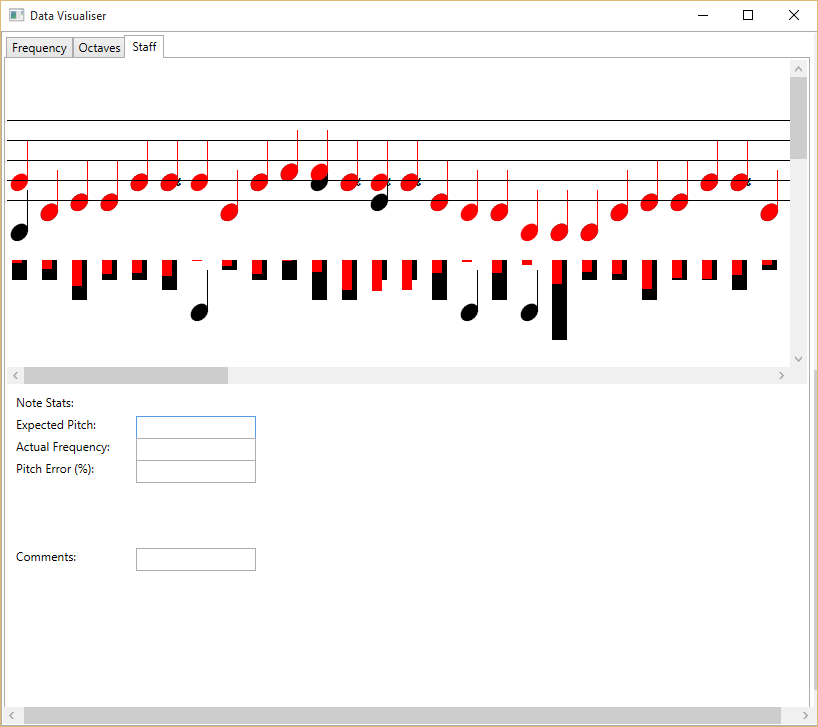
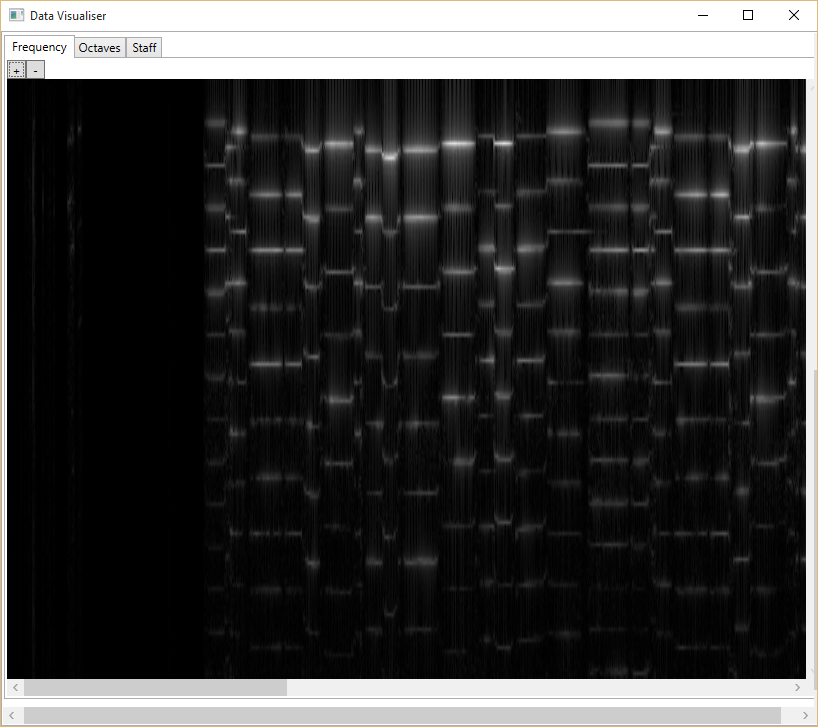
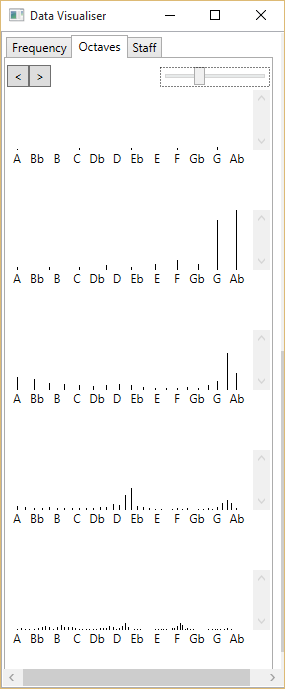
Fft() is called numerous times throughout the program and even calls itself to setup a quicksort-esc procedure. Each execution of the function takes around 4347ns to complete, and is referenced several thousand times through the music score. Reducing the execution time of this single function could provide significant savings over the many iterations. However, the recursive calling would have required extensive refactoring of the function and testing to ensure correctness. Due to time constraints this was not deemed worthwhile to pursue while parallelising other simpler structures would provide similar performance gains.

Overall, the 2.0-2.5x speedup of the application is noticeably significant when compared to the sequential program. Based on this speedup result, we consider this parallelisation successful, although there are further potential aspects of the program for parallelisation which could reduce the execution time further.

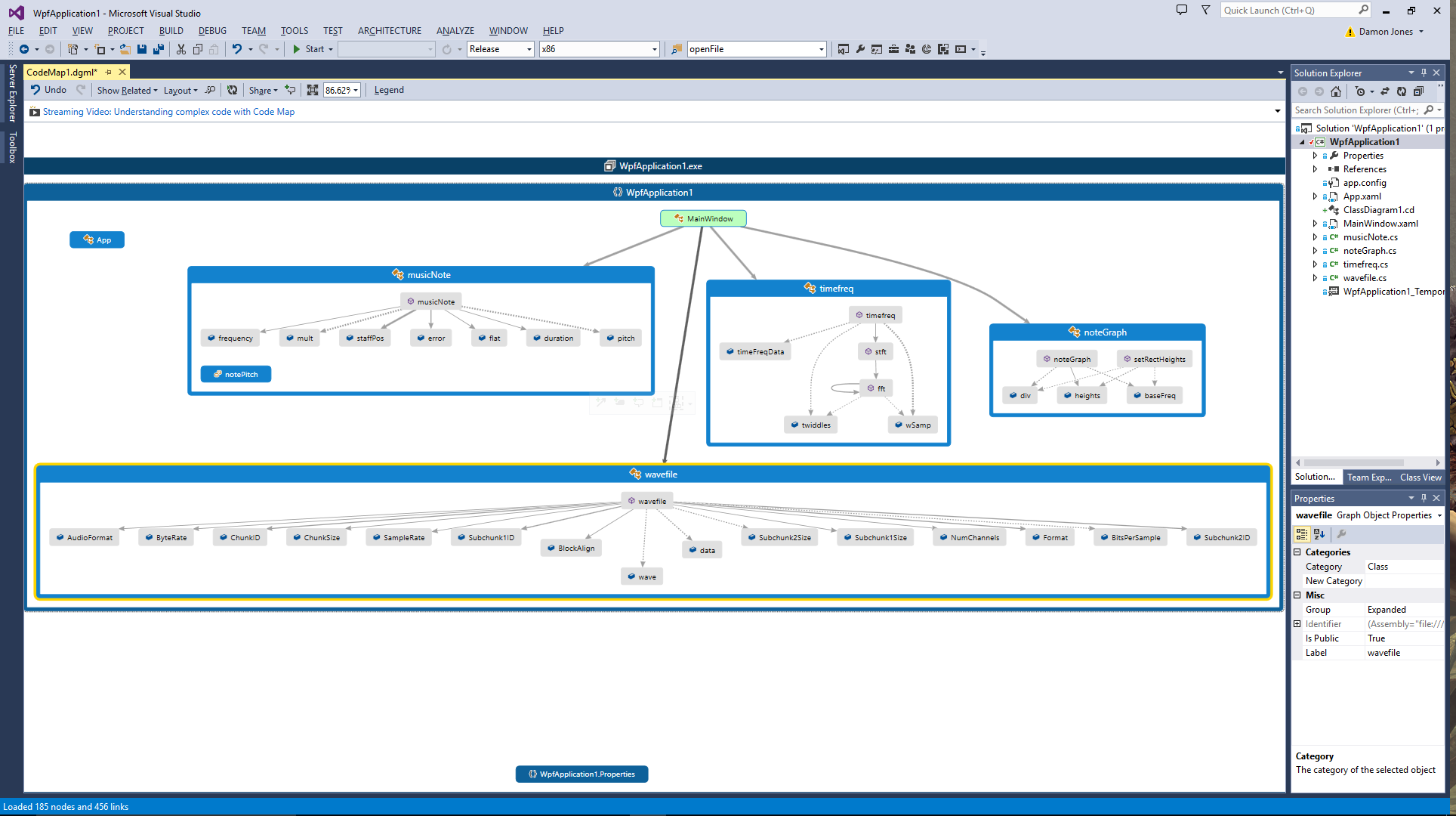
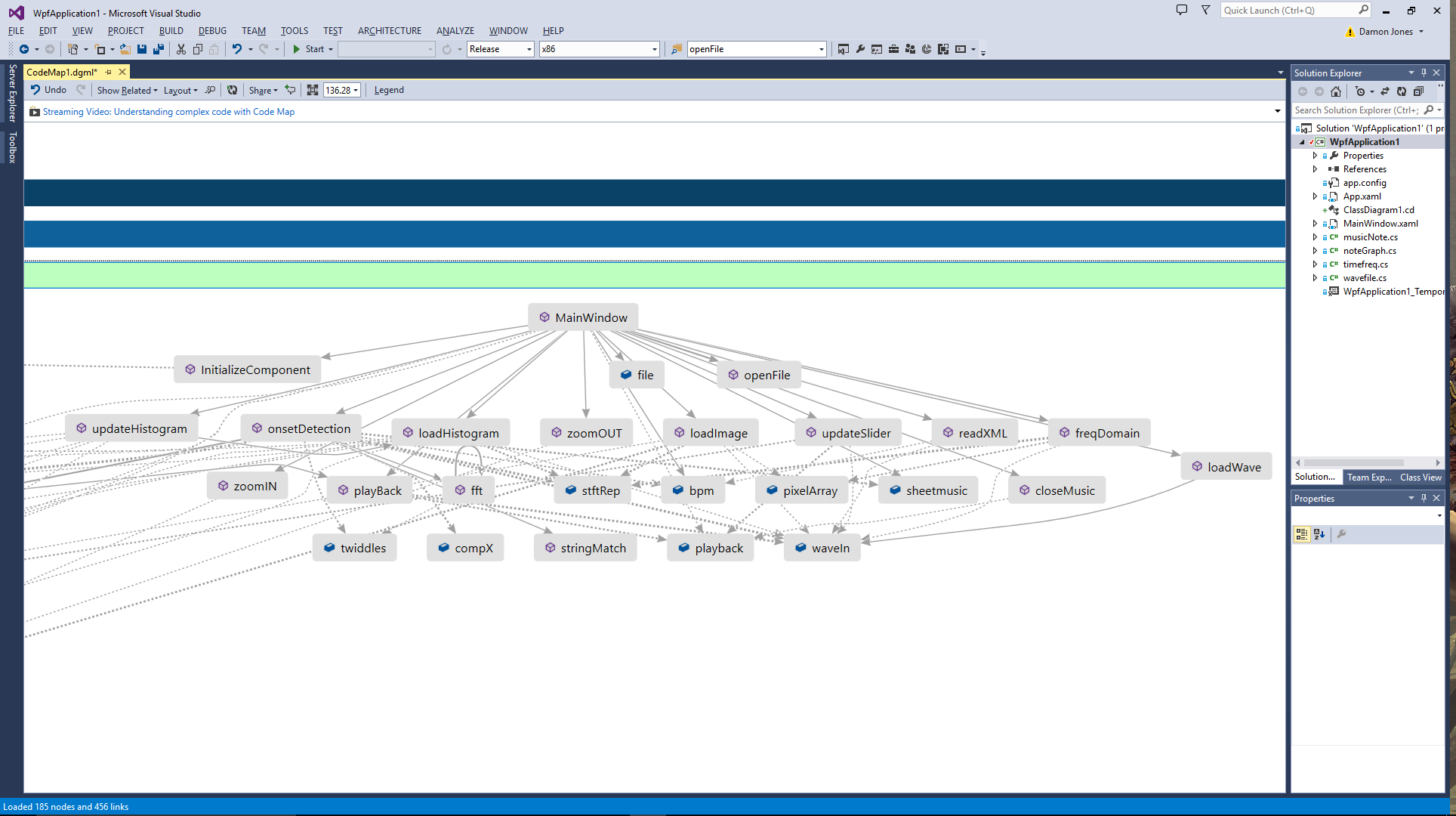
Thread pooling is another potential technique which would have been beneficial to implement. Thread pools would have reduced the overheads of creating threads and allowed for a faster execution time. Using the system environment variables to identify the number of cores available would also have been advantageous to implement instead of the manually specific max\_threads constant.

# Appendix

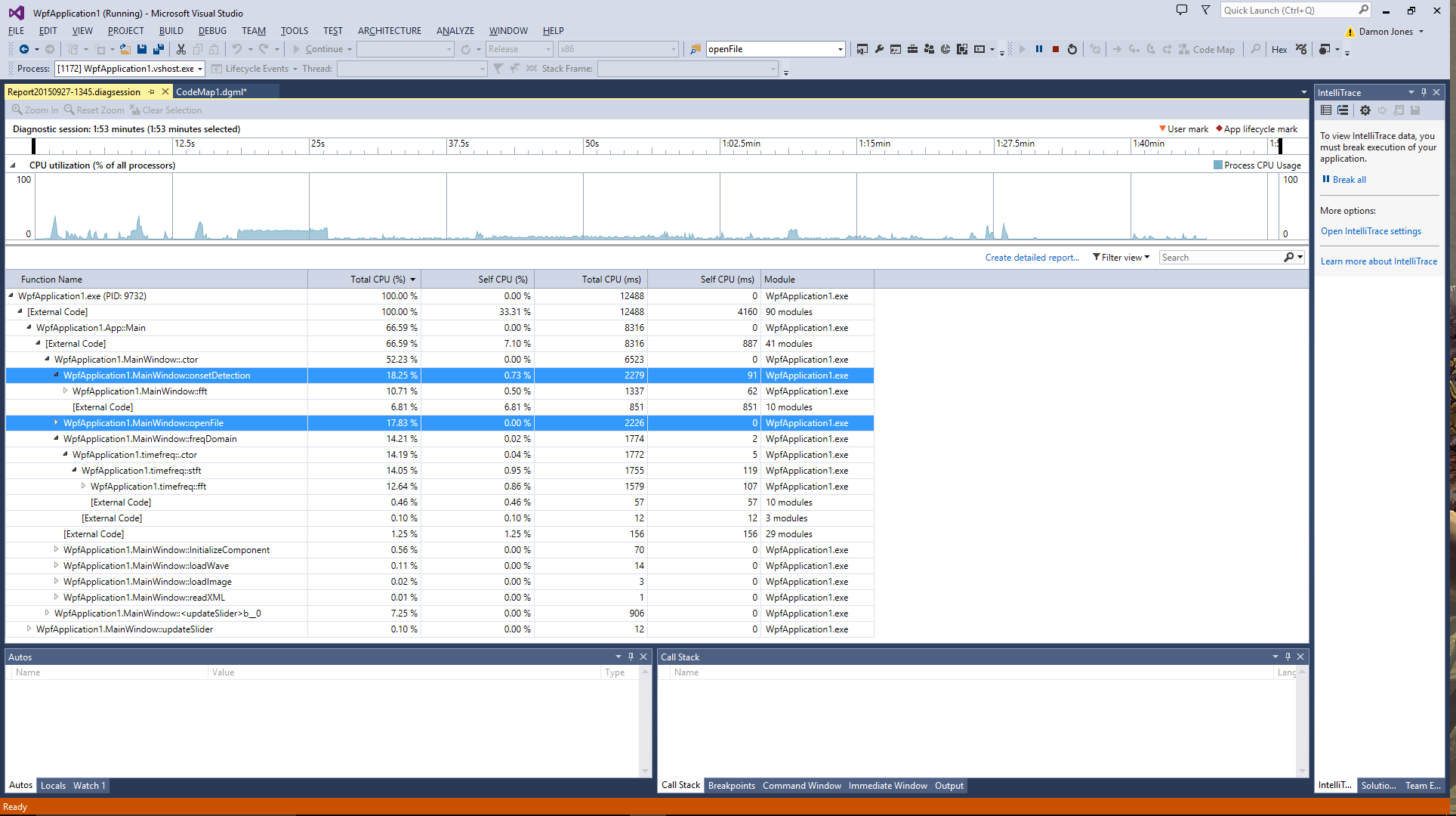
## Digital Music GUI



## Class diagram



## Visual Studio Diagnostic Session



## Sequential Timings

### Overall Timings

Load Audio:

208ms

208700464ns

slider thread:

0ms

34928ns

loadWave:

142ms

142183864ns

freqDomain:

12024ms

12009076176ns

sheetmusic:

5ms

5922656ns

onsetDetection:

10490ms

10477279944ns

loadImage:

13ms

13491648ns

loadHistogram:

18ms

18242328ns

playBack:

670ms

669962936ns

Other stuff:

1ms

1797848ns

### FreqDomain()

timefreq - 1:

0ms

864087ns

timefreq - 2:

39ms

38981481ns

timefreq - 3:

5ms

5622603ns

timefreq - stft:

5476ms

5410643763ns

freqDomain timefreq:

5525ms

5458631745ns

freqDomain pixelarray:

0ms

23667ns

freqDomain loop:

25ms

25078809ns

### OnsetDetection()

Onset setup:

0ms

49749ns

Onset loop 1:

276ms

273529179ns

Onset loop 2:

0ms

291249ns

Onset loop 3:

0ms

14007ns

Onset loop 4:

0ms

2898ns

onset mm loop:

5981ms

5909716554ns

onset loop 6:

0ms

764106ns

onset loop 7:

0ms

3864ns

onset loop 8:

0ms

1932ns

onset stuff:

1ms

1793379ns

onset loop 9:

0ms

20769ns

onset loop 10:

5ms

5641923ns

onset loop 11:

6ms

6894342ns

onset loop 12:

3ms

3777543ns

onset loop 13:

3ms

3847095ns

### mm loop

Onset mm - ll:

7ms

7115556ns

Onset mm - kk:

0ms

545307ns

Onset mm - Y = fft:

27ms

26705553ns

Onset mm - jj:

1ms

1088199ns

Onset mm - div:

0ms

2898ns

Onset mm - last:

0ms

1932ns

### Timefreq()

timefreq - 1:

1ms

1188663ns

timefreq - 2:

58ms

58229031ns

timefreq - 3:

14ms

14691894ns

timefreq - stft:

3741ms

3696058485ns

### Timefreq – stft()

timefreq@stft - 1:

42ms

41536551ns

timefreq@stft - 2:

3640ms

3597110139ns

timefreq@stft - 3:

56ms

55339725ns

## Development per Loop Timings

### Timefreq loop 2, Timefreq stft loop 2, Timefreq stft loop 3, onSetDetection loop 1, onSetDetection mm loop

This is the main loop of onsetDetection

Normal

---------------------------------------

onset mm loop:

5798ms

5728875075 ns

onset mm loop:

5581ms

5514457851 ns

onset mm loop:

5629ms

5561997126 ns

Threaded

---------------------------------------

onset mm loop:

2569ms

2538210885 ns

onset mm loop:

2661ms

2629436544 ns

onset mm loop:

2594ms

2563488207 ns

Average

---------------------------------------

N: (5728875075 + 5514457851 + 5561997126)/3 = 5601776684

P: (2538210885 + 2629436544 + 2563488207)/3 = 2577045212

S^ = n/p = 5601776684/2577045212 = 2.1737207628

This is the final modification of Y in stft() in timefreq

Normal

---------------------------------------

timefreq@stft - 3:

73ms

72509892ns

timefreq@stft - 3:

72ms

71745786ns

timefreq@stft - 3:

73ms

72124458ns

Threaded

---------------------------------------

timefreq@stft - 3:

27ms

27119967ns

timefreq@stft - 3:

30ms

30034389ns

timefreq@stft - 3:

23ms

23488290ns

Average

---------------------------------------

N: (72509892 + 71745786 + 72124458)/3 = 72126712

P: (27119967 + 30034389 + 23488290)/3 = 26880882

S^ = n/p = 72126712/26880882 = 2.68319737425

This is the timefreq stft section 2

Normal

---------------------------------------

timefreq@stft - 2:

6618ms

6538439586 ns

timefreq@stft - 2:

6866ms

6784103046 ns

timefreq@stft - 2:

6497ms

6418973883 ns

Threaded

---------------------------------------

timefreq@stft - 2:

2053ms

2029218240 ns

timefreq@stft - 2:

2023ms

1998659796 ns

timefreq@stft - 2:

2159ms

2133680997 ns

Average

---------------------------------------

N: (6538439586 + 6784103046 + 6418973883)/3 = 6580505505

P: (2029218240 + 1998659796 + 2133680997)/3 = 2053853011

S^ = n/p = 6580505505/2053853011 = 3.20398074729

This is the initial population of compX

Normal

---------------------------------------

timefreq - 2:

44ms

44370312ns

timefreq - 2:

39ms

38981481ns

timefreq - 2:

34ms

34507935ns

Threaded

---------------------------------------

timefreq - 2:

17ms

16966341ns

timefreq - 2:

17ms

17325210ns

timefreq - 2:

27ms

26818575ns

Average

---------------------------------------

N: (44370312 + 38981481 + 34507935)/3 = 39286576

P: (16966341 + 17325210 + 26818575)/3 = 20370042

S^ = n/p = 39286576/20370042 = 1.92864482066

This is the 1st loop of onsetDetection. Populates HFC

Normal

---------------------------------------

Onset loop 1:

272ms

269340603 ns

Onset loop 1:

273ms

269729901 ns

Onset loop 1:

302ms

299239269 ns

Threaded

---------------------------------------

Onset loop 1:

83ms

82677525 ns

Onset loop 1:

50ms

49935921 ns

Onset loop 1:

77ms

76805694 ns

Average

---------------------------------------

N: (269340603 + 269729901 + 299239269)/3 = 279436591

P: (82677525 + 49935921 + 76805694)/3 = 69806380

S^ = n/p = 279436591/69806380 = 4.00302366345