Parallel Project – Digital Music Analysis

CAB401

Matthew Topping
N9713344

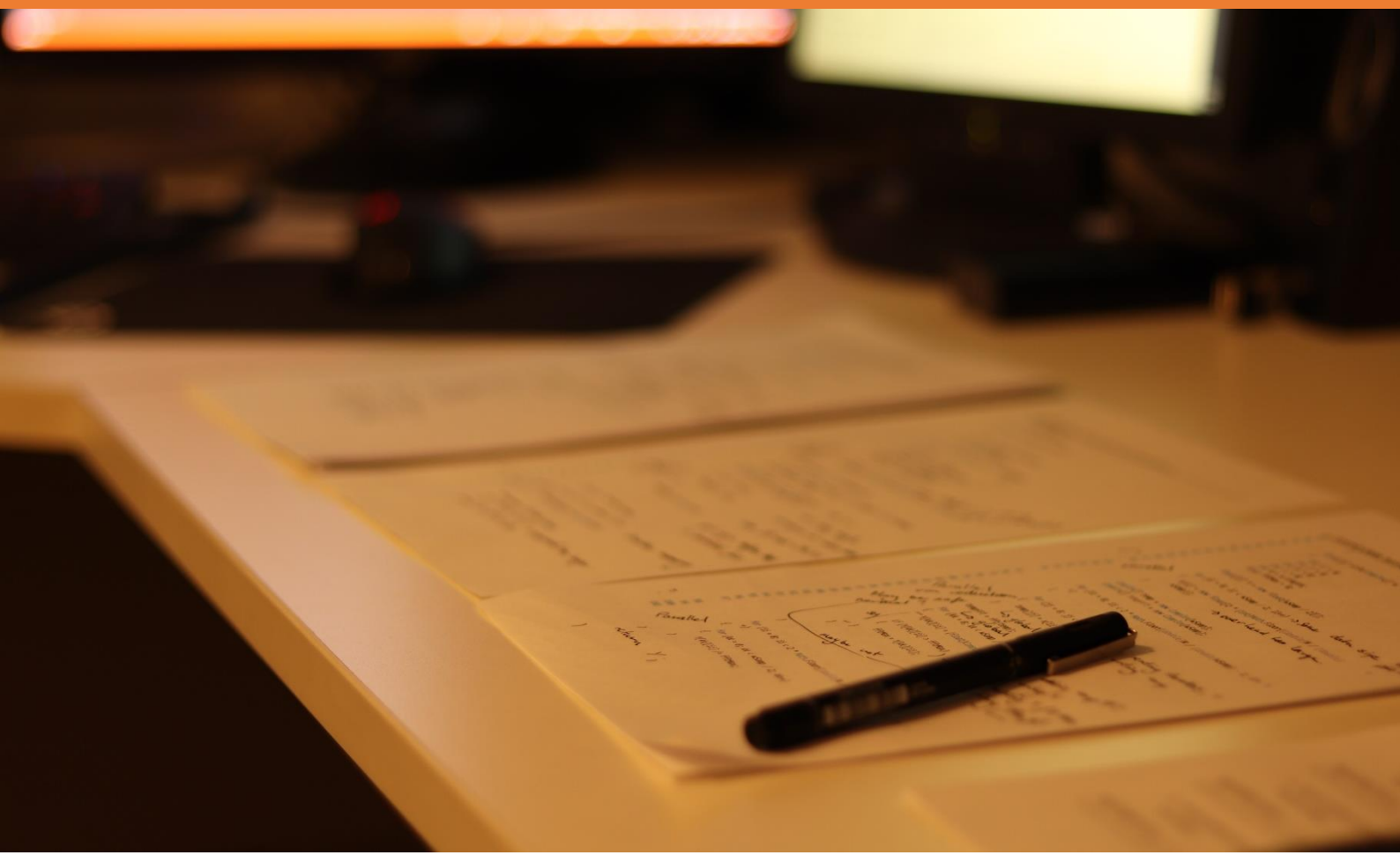


Table of Contents

1.0 The Application

The application selected is a music analysis program, designed to listen to a recording and provide feedback to the player. For a beginner learning a fretless instrument like one from the violin family, it can be hard to determine whether the notes being played are the correct pitch as there is no hardware guide to inform the player where certain notes are played on the instrument. There is line of products for violins (example in Appendix 1) that place fret guides on the instrument to help players learn these placements, however these may not assist in a player hearing what notes are correct. This program aims to bridge this gap.

The program takes a recorded wave file of the player and identifies whether the notes played are matched to a provided score. This looks at both pitch and note length and provides the user feedback through a GUI. To achieve this, the program executes a series of Fourier transforms. This process analyses the digital audio waveform and interprets the collection of harmonics that an instrument naturally makes to determine the frequency with the largest amplitude. This frequency is then determined as the note.

Looking into the details, the program executes 3 key stages. Firstly, the program executes a collection of short-term Fourier transforms to get pitches across the program for fractions of a second. Once this is complete, the program determines where notes begin and end. This is done through on set detection and uses the data generated from the execution of all the short-term Fourier transforms to determine where notes start and end. Once the program knows the start and end of a note, the program re-executes a Fourier transform across the whole note allowing the information to be averaged giving a more precise indication of the note played over that interval.

The program executes this process under a collection of calls between two classes and several functions. To summarise Appendix 2, the program initially reads in the audio file and XML file. Once this is done, the program calls freqDomain(). This is a private function in the MainWindow class that creates a new timefreq object that will do the initial Fourier transforms. Timefreq class converts the wave data in a complex form to allow for Fourier transforms to take place. The timefreq constructer then calls stft() which divides the audio into the small samples that will be analysed. These samples and then executed by fft() which does the Fourier transform computation recursively. Once this is done the timefreq object is constructed and contains the results of the short-term Fourier transforms. As freqDomain() updates a global class variable, onsetDetection() now have the data to do the next step of the analysis.

OnsetDetection() reads stftRep and executes the onsetDetection to determine the start and end point of each note. Once it has done so, it then calls a second local version of fft() to re-execute the Fourier transforms on the full length note. Once this is complete, the program will determine which octave the note is strongest in and adds it to a list of pitches ready to be displayed.

2.0 Parallelism Analysis

In order to achieve maximum parallel performance, it was clear to ensure that fft() calls where executed in a parallel manner. This is the case as the function does a large amount of the heavy lifting when doing the computation. Appendix 3 shows the profiling that was conducted to lead to this conclusion and it is seen that in both freqDomain() and onsetDetection(), fft() does the majority of the work.

In combination with this, some initial Parallel loops were added to non-nested loop segments. This identified that on average, parallel loops ran 6 times slower due to overhead as the amount of work these functions were doing was not enough to warrant parallelism. As a result, 4 key areas where Identified to be executed in parallel, two of which contain fft() as an inner call.

2.1 Timefreq Parallel Analysis

Appendix 4 highlights the process used to assess the sections of code that where parallelisable in timefreq. The nested for loop found on line 74 had a collection of key dependencies that had to be redesigned to allow the code to run in parallel. Firstly, all iterators were globally defined at the start of stft(). To remove these, all loops were written to contain the integer declaration. Furthermore, temp[] and tempFFT[] were also referring to global variables. To ensure that temp[] and tempFFT[] were not being raced to by each thread, the arrays would be defined inside a parallel section. As the data is written into the final array Y, this process is safe as all data is transferred to a globally visible variable.

Outside of these global variables, the if guard on line 88 creates a potential race condition. If two threads had a successful match of the guard at the same time, it is possible that two writes could be queued next to each other. This has the potential for a smaller value to overwrite the fftMax variable. To remove this risk, and make the code section safe for parallelism, the code must run as a reduction. The ii iterator would be divided into chunks and each section would have it's on local copy of fftMax. Once the computation in a thread is done, the code will write its own copy of fftMax into an array. From there, the same process can be applied again on the new data set until the problem is reduced to the comparison of 2 values.

The second nested for loop was also analysed to see if there was any potential parallelism. Like before, this section of the code also had globally defined iterators, thus, these would have to be declared in the loop to remove the dependency. Apart from this, the data is safe to run

in parallel as there is only a single read and write, meaning that no matter what order the iterations are run in, the array will still be completely covered safely.

2.2 OnsetDetection Parallel Analysis

The next key section that contains calls to fft() is onsetDetection. A lot of this function is parallelisable but only two key areas had enough granularity to overcome the associated overhead that comes with parallel programming. Appendix 5 outlines the process that was undertaken to analysis data dependence.

Firstly, the for loop found on line 305 was investigated for its ability to be parallelised. As seen in the timefreq, loop iterators were defined globally creating an unnecessary data dependency. These would have to be declared to allow for the loop to be run in parallel. Furthermore, stftRep is not being written to elsewhere, meaning that all reads and writes in this loop are safe for parallel computation.

The major parallel improvement in this section however was found in the large nested for loop starting on line 357. This had a collection of key data dependencies that made computation in parallel unsafe, and as a result required some restructuring to make safe. Firstly, there were a collection of variables that would be raced to be written to, if not redeclared in appropriate places. This included, the ll iterator, twiddles, compX, Y and absY. Most of these variables are safe to move inside of the for loop as previously suggested with other computation areas, however this does not work for the twiddles array. Twiddles is an array that is declared globally in MainWindow. This facilitates the computation conducted by fft() as it reads this data. As a result, if twiddles was to be moved inside the onsetDetection() for loop, fft() would no longer be able to see twiddles data thus creating an error. To address this, fft() in MainWindow could be restructured to have a third parameter that passes in the twiddles data. As fft() is only called by onsetDetection(), this would not disrupt the code elsewhere and allow for twiddles to be declared in a parallel safe location. The final restructure that would be required to make this call safe is to do the data storage for pitches. In its current implementation, pitches are using a list. As a result, threads can add to the list at anytime in any order. This means that the pitches order would not be retained if called in parallel. Pitches is defined with a set size of 100. This means that the data structure can be changed to an array with the same defined size. This would facilitate the ordering of notes and allow the code to be safely run in parallel.

2.3 Further Parallel Analysis

Several further areas were identified to be safe for parallelism through the program. None of these will be implemented however as all identified areas do not have enough granularity to warrant parallelism. This was confirmed by the implementation of such sections being timed and found to have a negative effect on performance due to overhead.

3.0 Parallel Implementation Tools and Techniques

To implement the parallel section, the Task Parallel Library was used. All parallel calls where made in an implicit nature using the higher level Parallel.For which abstracts a large amount of the threading that lower level constructs may use. Despite this, the program was written to both the base implicit Parallel.For calls and a block implementation. This allowed for the number of Parallel.For loop iterations to be predefined to better match the number of cores that the CPU was using to execute the program.

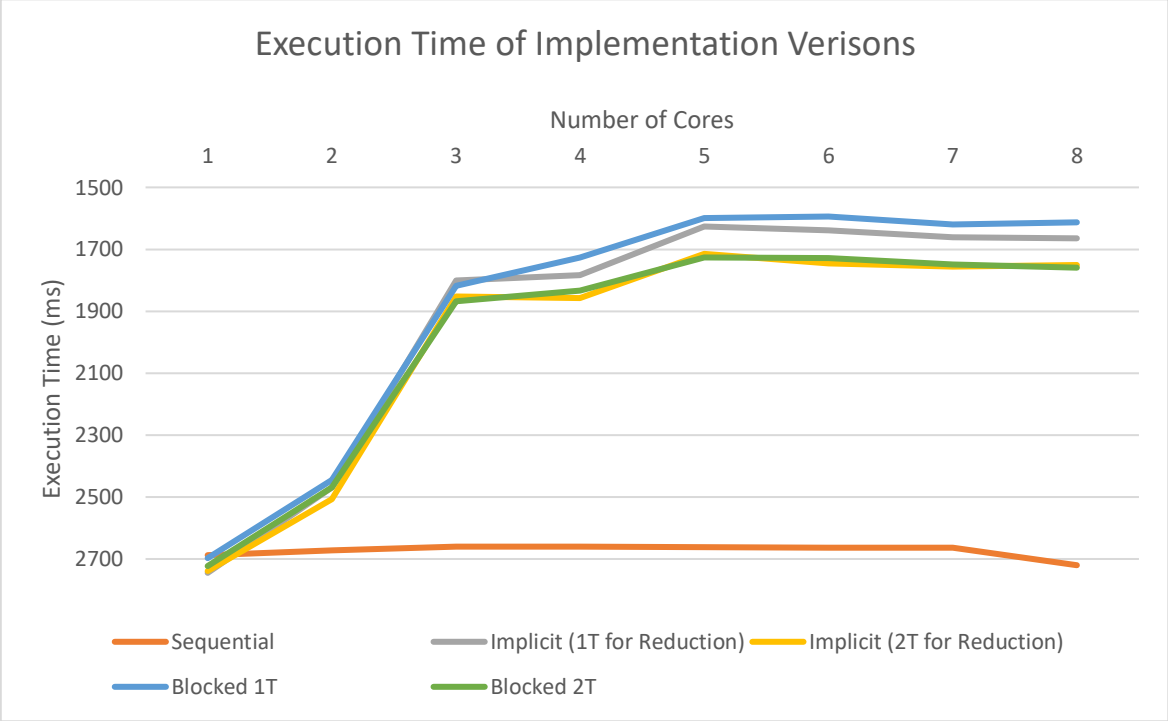
During testing, the global NUM_THREADS_USED in both timefreq and MainWindow would be matched to the number of cores. Further tests also included the value being doubled to see if more threads would create a different result. This implementation is more likely to effect load balancing across threads, so all results were recorded to see which of the parallel implementations was most effective for the Task Parallel Library.

Throughout the project a collection of key software and techniques were used to create the parallelised result. Visual Studio was used to do all programming, timing, profiling and compiling. As a result, this software played a critical role in achieving the solution. Beyond Visual Studio, task manager was used during the timing of solution. Task manager was used to “set affinity” of the application process restricting it to use only a set number of cores on the CPU. This was essential when measuring the speed up that was created by the parallel sections.

In order to implement the parallel sections a collection of techniques was used. Firstly, node analysis was conducted when dependencies were not clearly visible. This helped visualise what data was connected to what and whether sections of code were safe to call in parallel. During implementation, both reduction and separation was required. Reduction was used to address a race condition which could be solved by using it’s divide and conquer methodology. Separation was used when certain code elements had to be removed from a loop to make it safe. This was used to remove variable dependencies. Finally, restructuring was used to allow for code to be moved to again facilitate parallelism. Examples of this include the passing of data through a parameter instead of using globalised variables.

4.0 Parallel Results

After implementing each version of the parallel program, timing results were recorded over a 50-iteration average. The sequential program was recorded first, followed by the implicit calls with the reduction block being run with 1 and 2 threads per core and finally the fully blocked implementation with 1 and 2 threads per core.

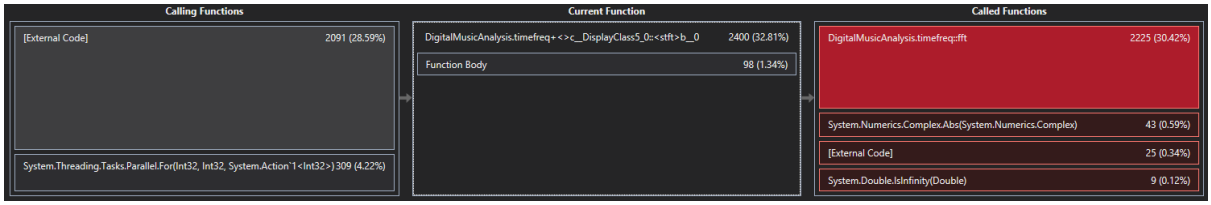


Graph 1: Execution time for all implementations

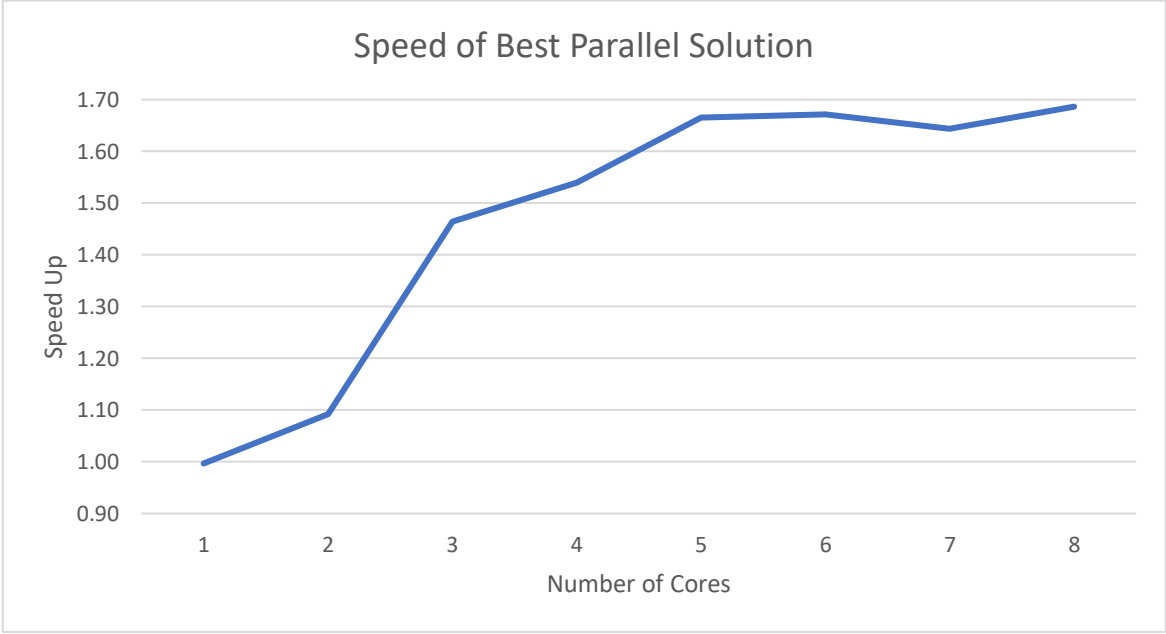
The results can be seen in Graph 1 and identify a few key trends. Firstly, as expected, the sequential program does not speed up when increasing the number of cores for execution. This sees the main bulk of computation take around 2700ms. Secondly, block numbers equivalent to the number of cores is faster than doubling the blocks. This is likely as a result of the increased over head that is required when creating these extra blocks and executing them in parallel. Finally, all parallel implementation follows similar speed up trends.

The optimum solution for the program was the blocked implementation with 1 block per core. This was seen to have a peak performance of 1594ms occurring on 6 cores as seen in Appendix 6. Despite this, execution time plateaued at 5 CPU cores. This means that the 16ms variation between the final four cores is likely to be due to other environmental factors outside of the program's execution.

Profiling between the two implementations highlights the program bottle neck. Appendix 3 shows the depth the fft() calls were abstracted by. In the image below, it can now be seen that fft() calls account for a larger amount of the CPU percentage and are appearing at a much higher level. This suggests that the parallel implementation has sped up the framework around the fft() calls effectively. To achieve further speed up, fft() would have to be investigated to increase performance further.



The speed up for the optimal solution follows a sub-linear trajectory (seen in Graph 2). It can also be seen that the solution provides a constant amount of parallelism as the speed up did not scale up past the fifth core. A likely cause of this kind of speed up curve is outlined by Amdahl's law (see Appendix 7). The law states that with a certain percentage of the program running in parallel, you return a certain amount of speed up. Based on Amdahl's estimations, it is likely that the optimal solution runs less than 50% of the code in parallel, resulting in the optimal speed up of 1.69 (see Appendix 8).



Graph 2: Speed up for optimal solution

5.0 Explanation of Parallel Adaptions

5.1 timefreq Parallel Implementation

First, the changes to the Parallel section found in timefreq at line 95 will be discussed. This is the reduction that finds the max value as a result of a call of fft(). The implementation is largely like the sequential implementation aside from a few key areas. Both temp[] and tempFFT[] are moved inside of any for loop operations. This removed the dependence that will occur when all threads refer to the same array in shared memory. On top of this, new data structures of threadMaxes and fftMax_thread are added on line 95 and 101 respectively. These are used for the reduction process. Finally, all iterators were declared in the loop to remove any dependencies.

This section of code executes the same, but across a number of blocks until reaching line 123. The program now writes to the new local fftMax_thread variable when a new max is found. At the end, the program then adds this value to a list instead of saving it to the final value. This allows each thread to find the max for its segment and then pass it for later sorting. The program then executes .Max() on the list to find the largest value and adds it to the original global max.

It is important to note that the implementation is not a true reduction. If the block size was to increase to account for a larger amount of cores (say beyond 32 cores), .Max() would run in its sequential manner on a much larger list. This could affect the scalability of this section of code, however it serves its larger purpose by calling fft() in parallel.

Finally, line 142 sees the implementation of a nested loop in parallel. This loop is internally the same with a blocked Parallel.For around it. Like previously, all iterators were called in the loop to remove any unnecessary dependencies. No extra techniques were required to run this loop in parallel.

5.2 MainWindow Parallel Implementation

Within MainWindow, three areas of code were changed to make two areas parallel. Firstly, a Parallel.For was implemented on line 313. This piece of code is a simple loop parallelisation with no internal changes except for the removal of all iterator declarations that occur outside of the loops. All variables created are there to create the block sizes used for execution.

By far the largest parallel implementation comes from the parallelisation of the nested for loop at line 373. Firstly, twiddles, compX, Y and absY have all moved their declaration and instantiation to inside the parallel section. The code then executes the original sequential code in parallel until reaching line 414. At this point the new parameter for fft() is added passing in the value of twiddles for computation. Now switching focus to line 749 fft(), a new parameter called fft_twiddles is added into the function declaration. This is then used for the computation, instead of the previous globally defined twiddles. Line 781 and 782, show that the recursive calls also must be updated to facilitate this change. Each recursive call will now be able to see twiddles as it passes the data through to itself each time.

Finally, looking at code lines 449 and 453, pitches has been converted to an array and each pitch is now added based on an index. This stops the unordering of notes that occur due parallel execution. The pitches array is defined on line 293 with the same size as the original list in the sequential program.

6.0 Project Difficulties

A large amount of the project difficulties came from data dependencies. This would see unexplainable errors occur which couldn't be debugged through a normal process. As a result new approaches were found to tackle the problems.

The first issue in relation to data dependencies saw iterators seemingly skipping indexes and not filling out arrays correctly. This was seen in the implementation of the parallel sections in timefreq at line 98. When implementing the Parallel.For the internal iterators would execute normally but the Parallel.For would skip indexes. This occurred in both the blocked and implicit implementations.

Initially, the problem was addressed via normal debugging process. The iterator was printed out to see if there was any consistent pattern to the skipped iterators to ensure the loops were written correctly. Furthermore, the size of the final maximum value of the iterator was checked to ensure the loop guard wasn't flawed. All of these returned no further clues as to way the iterator was skipping. Ultimately, the problem was solved by going back to the drawing board. The original sequential implementation of the code was re-analysed to see if there were any missed data dependencies. It was found temp[] and tempFFT[] were defined outside of the loops and as a result, each thread was writing and reading from the same shared temporary memory slot. By addressing this and moving the variables back into a thread safe position, the section could be implemented dramatically improving performance.

Similarly, data dependencies were causing index out of range errors whilst implementing the parallel section in MainWindow at 373. Once again, the error was ambiguous as executing the code in parallel caused an index out of range error in an array associated with the function displaying information to the GUI. This was a bit more obvious that a race condition was the culprit, as arrays were being written to in an uncontrolled manner. To solve this, a deeper look at the large nested for loop was undertaken and it was identified that twiddles was causing the issue. As discussed earlier, this was resolved by restructuring the code to remove the data dependency by passing the data locally in a thread through function parameters. This facilitated the parallelisation of the outer loop which dramatically increased performance.

Finally, granularity caused a lot of slowdowns. Throughout the code, certain single for loops were implemented in parallel. These were always found to run slower due to the overhead that is created. To resolve this issue, these sections were simple removed.

7.0 Reflection

This project has been an eye-opening experience into some of my shortfalls as a programmer. I am now aware that a lot of my knowledge is based on higher level programming and my knowledge on the specific of things such as memory and hardware are limited. As a result, the time spent on the project was used to broaden my knowledge on these topics rather than applying it to the program.

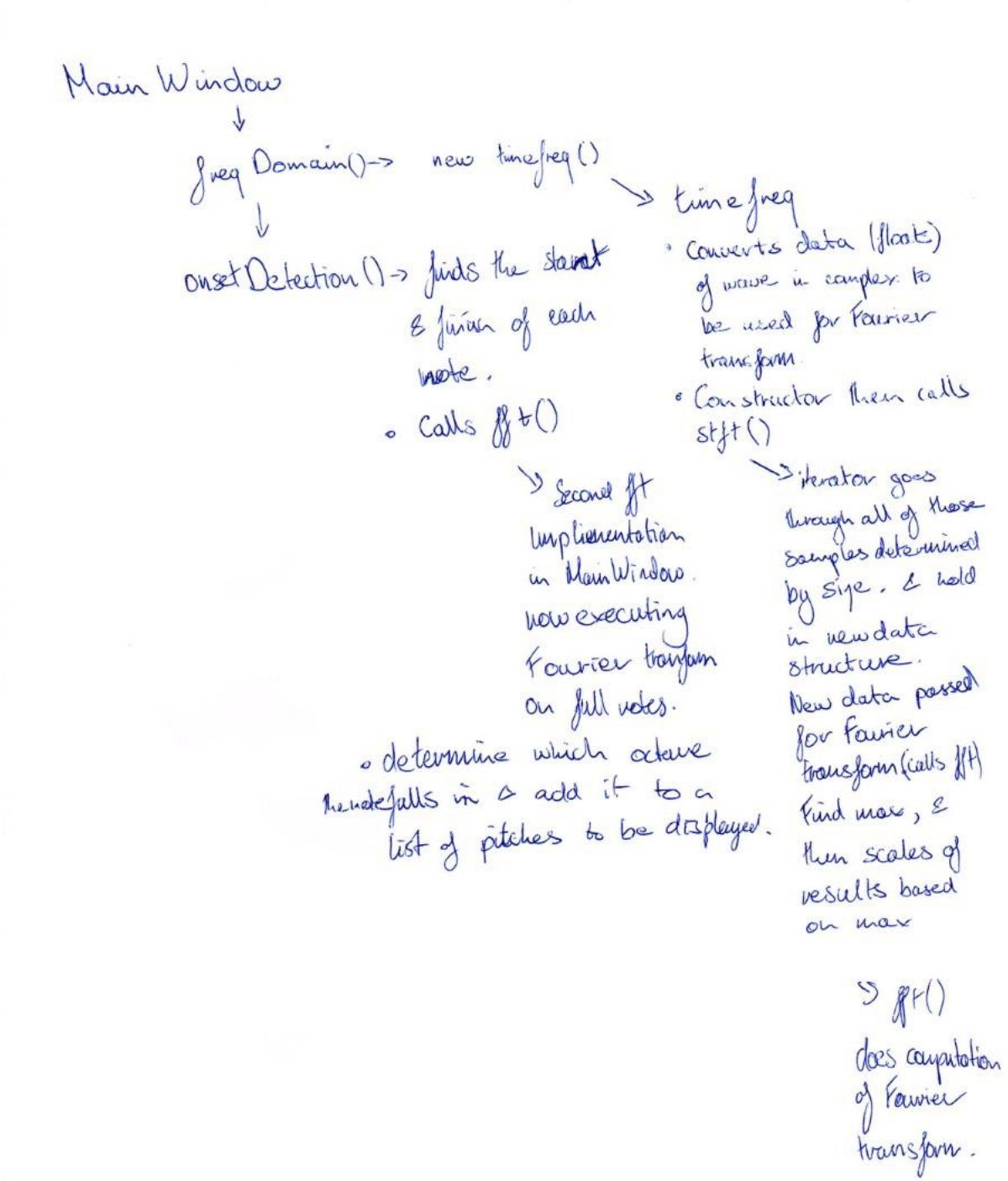
Overall, this means that I think that this solution is not the optimal. The solution overlooks speedup techniques like array flattening to increase cache locality and specific complier settings specific to my CPU. Despite this, I believe that the attempt is reasonable. Although the attempt did not get full scalability up to 8 cores, it did see it past 4. Furthermore, the parallel sections implemented targeted the key computationally heavy section of the program. These were achieved through more than just a Parallel.For on a bit of code which shows the improvement of my understanding of data dependencies throughout the project.

Finally, there is a collection of things I could have done differently. Firstly, I could have spent more time in the profiler to find other computationally heavy parts of the code. This would have allowed for further speedup. To second this, I would investigate control dependencies. Fft() and its recursive nature meant that I did not consider it for parallelisation due to the control dependencies. As this was computationally heavy, I would next time like to investigate whether it could be re-written into a thread safe iterative version to allow for parallelisation and increased performance. Finally, I would choose a program in a difference language that allowed me to use explicit threads. This would allow me to code at a lower level helping me understand a lot more of the fundamentals of parallel programming, hopefully resulting in an improved solution.

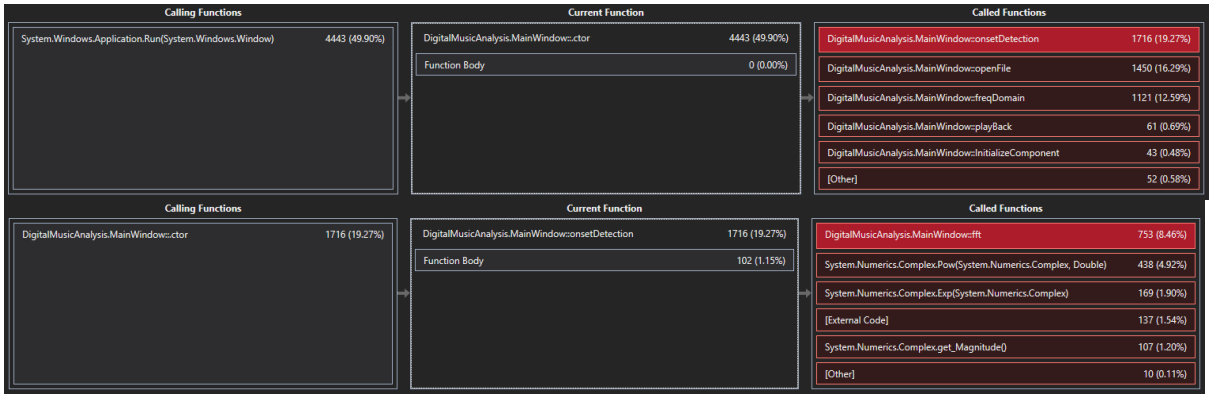
8.0 Appendices

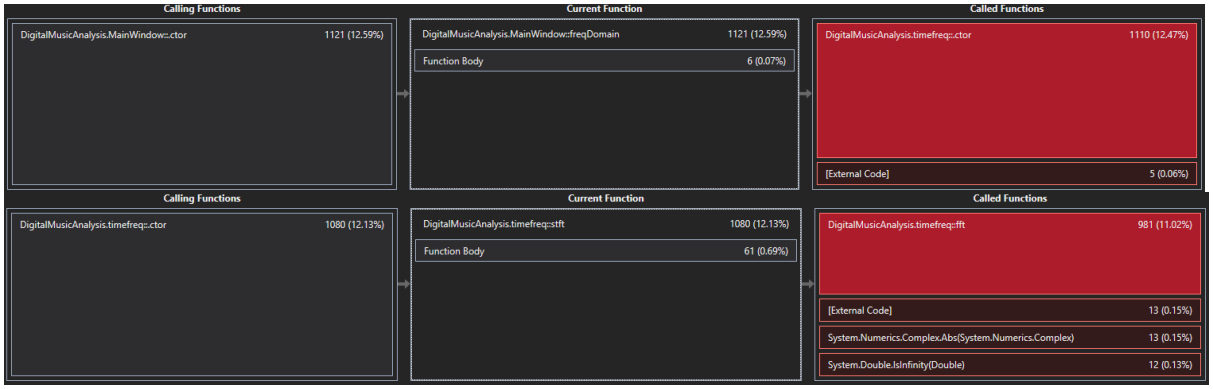
Appendix 1: Fret guides for violin players

Appendix 2: Call Outline for the Music Analysis Program

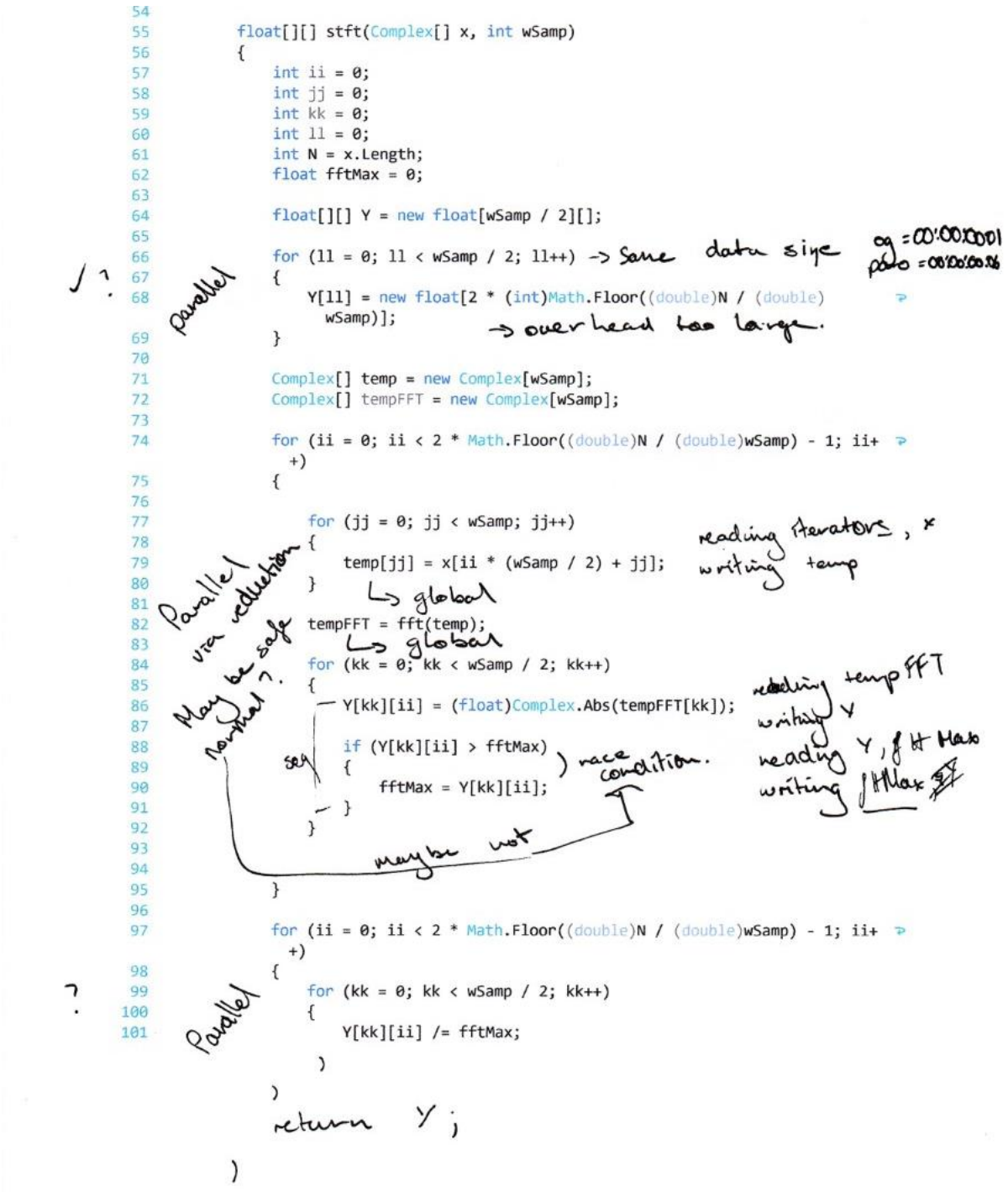


Appendix 3: Profiling for fft()





Appendix 4: timefreq analysis

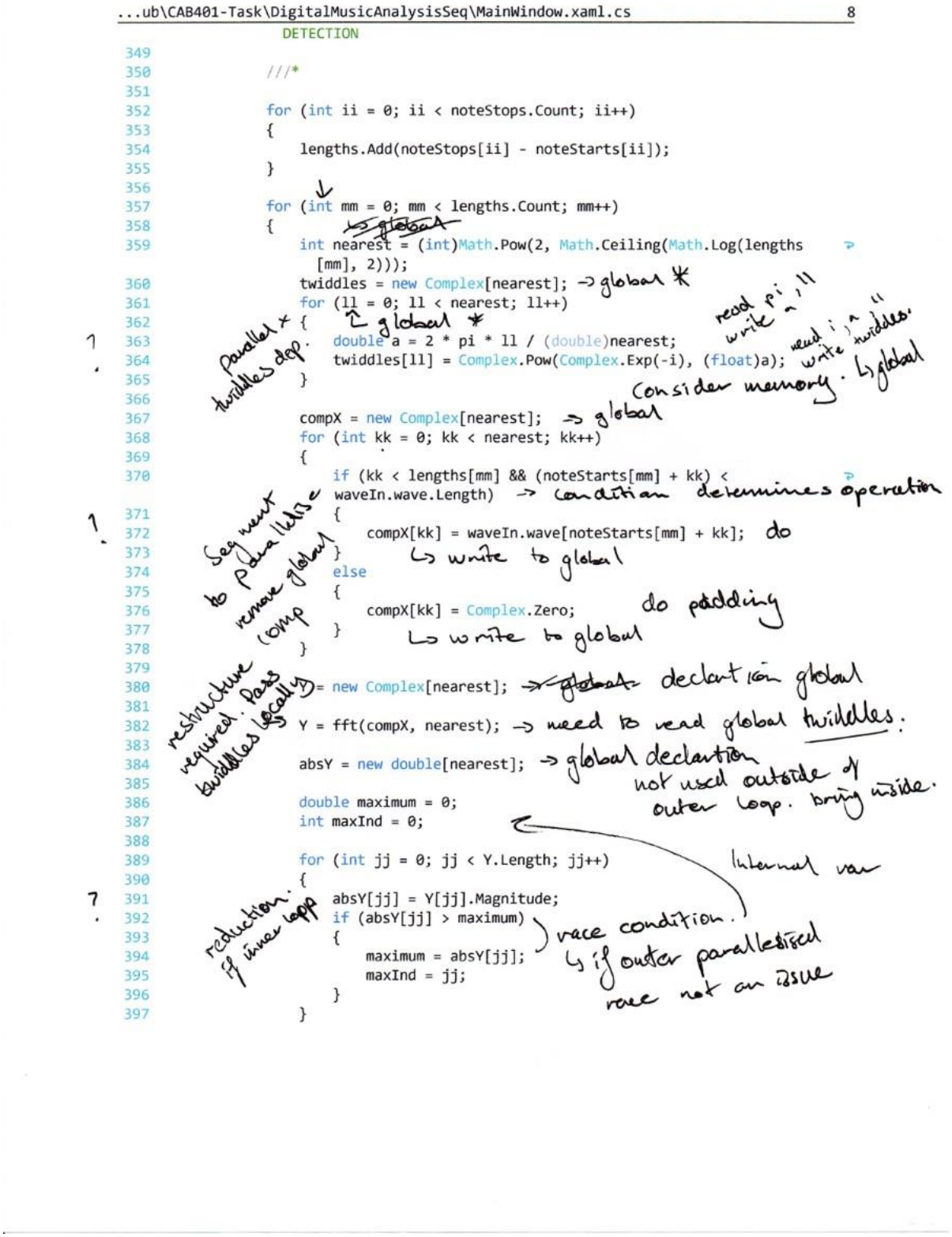


```
54
55      float[][] stft(Complex[] x, int wSamp)
56      {
57          int ii = 0;
58          int jj = 0;
59          int kk = 0;
60          int ll = 0;
61          int N = x.Length;
62          float fftMax = 0;
63
64          float[][] Y = new float[wSamp / 2][];
65
66          for (ll = 0; ll < wSamp / 2; ll++)
67          {
68              Y[ll] = new float[2 * (int)Math.Floor((double)N / (double)
                  wSamp)];
69          }
70
71          Complex[] temp = new Complex[wSamp];
72          Complex[] tempFFT = new Complex[wSamp];
73
74          for (ii = 0; ii < 2 * Math.Floor((double)N / (double)wSamp) - 1; ii+
                +)
75          {
76
77              for (jj = 0; jj < wSamp; jj++)
78              {
79                  temp[jj] = x[ii * (wSamp / 2) + jj];
80              }
81
82              tempFFT = fft(temp);
83
84              for (kk = 0; kk < wSamp / 2; kk++)
85              {
86                  Y[kk][ii] = (float)Complex.Abs(tempFFT[kk]);
87
88                  if (Y[kk][ii] > fftMax)
89                  {
90                      fftMax = Y[kk][ii];
91                  }
92              }
93
94
95          }
96
97          for (ii = 0; ii < 2 * Math.Floor((double)N / (double)wSamp) - 1; ii+
                +)
98          {
99              for (kk = 0; kk < wSamp / 2; kk++)
100             {
101                 Y[kk][ii] /= fftMax;
                )
            )
            return Y;
        )
```

Appendix 5: onsetDetection() analysis

...ub\CAB401-Task\DigitalMusicAnalysisSeq\MainWindow.xaml.cs 7

```
298
299             SolidColorBrush sheetBrush = new SolidColorBrush(Colors.Black);
300             SolidColorBrush ErrorBrush = new SolidColorBrush(Colors.Red);
301             SolidColorBrush whiteBrush = new SolidColorBrush(Colors.White);
302
303             HFC = new float[stftRep.timeFreqData[0].Length];
304
305             for (int jj = 0; jj < stftRep.timeFreqData[0].Length; jj++)
306             {
307                 for (int ii = 0; ii < stftRep.wSamp / 2; ii++)
308                 {
309                     HFC[jj] = HFC[jj] + (float)Math.Pow((double)
                        stftRep.timeFreqData[ii][jj] * ii, 2);
310                 }
311
312             }
313
314             float maxi = HFC.Max();
315
316             for (int jj = 0; jj < stftRep.timeFreqData[0].Length; jj++)
317             {
318                 HFC[jj] = (float)Math.Pow((HFC[jj] / maxi), 2);
319             }
320
321             for (int jj = 0; jj < stftRep.timeFreqData[0].Length; jj++)
322             {
323                 if (starts > stops)
324                 {
325                     if (HFC[jj] < 0.001)
326                     {
327                         noteStops.Add(jj * ((stftRep.wSamp - 1) / 2));
328                         stops = stops + 1;
329                     }
330                 }
331                 else if (starts - stops == 0)
332                 {
333                     if (HFC[jj] > 0.001)
334                     {
335                         noteStarts.Add(jj * ((stftRep.wSamp - 1) / 2));
336                         starts = starts + 1;
337                     }
338
339                 }
340             }
341
342             if (starts > stops)
343             {
344                 noteStops.Add(waveIn.data.Length);
345             }
346
347
348             // DETERMINES START AND FINISH TIME OF NOTES BASED ON ONSET
```


...ub\CAB401-Task\DigitalMusicAnalysisSeq\MainWindow.xaml.cs 8
DETECTION
///*
for (int ii = 0; ii < noteStops.Count; ii++)
{
lengths.Add(noteStops[ii] - noteStarts[ii]);
}
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]; -> global *
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]; -> global
for (int kk = 0; kk < nearest; kk++)
{
if (kk < lengths[mm] && (noteStarts[mm] + kk) < waveIn.wave.Length) -> condition determines operation
{
compX[kk] = waveIn.wave[noteStarts[mm] + kk]; do
}
else
{
compX[kk] = Complex.Zero; do padding
}
}
= new Complex[nearest]; declaration global
Y = fft(compX, nearest); -> need to read global twiddles.
absY = new double[nearest]; -> global declaration
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;
}
}
parallel x twiddles dep.
read pi, ll write a
read i, a write twiddles.
Consider memory. -> global
Segment to parallelise remove global (comp
restructure required. pass twiddles locally
not used outside of outer loop. bring inside.
reduction if inner loop
race condition. if outer parallelised race not an issue

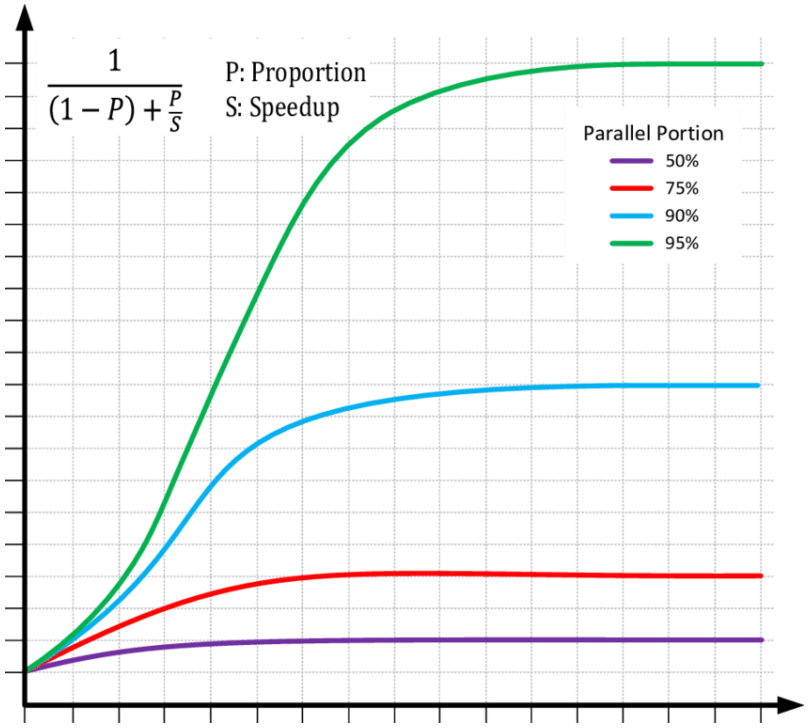
...ub\CAB401-Task\DigitalMusicAnalysisSeq\MainWindow.xaml.cs 9

```
398
399                 for (int div = 6; div > 1; div--)
400                 {
401
402                     if (maxInd > nearest / 2)
403                     {
404                         if (absY[(int)Math.Floor((double)(nearest - maxInd) /
                    div)] / absY[(maxInd)] > 0.10)
405                         {
406                             maxInd = (nearest - maxInd) / div;
407                         }
408                     }
409                     else
410                     {
411                         if (absY[(int)Math.Floor((double)maxInd / div)] / absY
                    [(maxInd)] > 0.10)
412                         {
413                             maxInd = maxInd / div;
414                         }
415                     }
416                 }
417
418                 if (maxInd > nearest / 2)
419                 {
420                     pitches.Add((nearest - maxInd) * waveIn.SampleRate /
                    nearest);
421                 }
422                 else
423                 {
424                     pitches.Add(maxInd * waveIn.SampleRate / nearest);
425                 }
426
427
428             }
429
430             musicNote[] noteArray;
431             noteArray = new musicNote[noteStarts.Count()];
432
433             for (int ii = 0; ii < noteStarts.Count(); ii++)
434             {
435                 noteArray[ii] = new musicNote(pitches[ii], lengths[ii]);
436             }
437
438             int[] sheetPitchArray = new int[sheetmusic.Length];
439             int[] notePitchArray = new int[noteArray.Length];
440
441             for (int ii = 0; ii < sheetmusic.Length; ii++)
442             {
443                 sheetPitchArray[ii] = sheetmusic[ii].pitch % 12;
444             }
445
446             for (int jj = 0; jj < noteArray.Length; jj++)
```

Appendix 6: Raw Execution Time Data

Number of Cores	Sequential	Implicit (Reduction 1T)	Implicit (Reduction 2T)	Blocked 1T	Blocked 2T
1	2688	2745	2739	2697	2723
2	2671	2469	2508	2445	2468
3	2661	1801	1853	1818	1867
4	2660	1783	1858	1727	1833
5	2662	1626	1715	1599	1726
6	2664	1639	1745	1594	1728
7	2663	1661	1756	1620	1749
8	2720	1664	1750	1613	1759

Appendix 7: Amdahl's Law



Appendix 8: Raw Speedup Results

Number of Cores	Speedup
1	1.00
2	1.09
3	1.46
4	1.54
5	1.67
6	1.67
7	1.64
8	1.69

Appendix 9: MainWindow Parallel Code Changes

```
95. List<float> threadMaxes = new List<float>();
96.
97. Parallel.For(0, NUM_THREADS_USED, iterator =>
98. {
99.       Complex[] temp = new Complex[wSamp];
100.      Complex[] tempFFT = new Complex[wSamp];
101.      float fftMax_thread = 0;
102.
103.      int guardSize = 2 * (int)Math.Floor(N / (double)wSamp) - 1;
104.      int chunkSize = (guardSize + (NUM_THREADS_USED - 1)) / NUM_THREADS_USED;
105.      int start = chunkSize * iterator;
106.      int end = Math.Min(start + chunkSize, guardSize);
107.
108.      for (int ii = start; ii < end; ii++)
109.      {
110.          for (int jj = 0; jj < wSamp; jj++)
111.          {
112.              temp[jj] = x[ii * (wSamp / 2) + jj];
113.          }
114.
115.          tempFFT = fft(temp);
116.
117.          for (int kk = 0; kk < wSamp / 2; kk++)
118.          {
119.              Y[kk][ii] = (float)Complex.Abs(tempFFT[kk]);
120.
121.              if (Y[kk][ii] > fftMax_thread)
122.              {
123.                  fftMax_thread = Y[kk][ii];
124.              }
125.          }
126.      }
127.      threadMaxes.Add(fftMax_thread);
128. });
129.
130. fftMax = threadMaxes.Max();
131. fftMax = threadMaxes.Max();
```

```
142.  Parallel.For(0, NUM_THREADS_USED, iterator =>
143.  {
144.      int guardSize = 2 * (int)Math.Floor(N / (double)wSamp) - 1;
145.      int chunkSize = (guardSize + (NUM_THREADS_USED - 1)) / NUM_THREADS_USED;
146.      int start = chunkSize * iterator;
147.      int end = Math.Min(start + chunkSize, guardSize);
148.
149.      for (int ii = start; ii < end; ii++)
150.      {
151.          for (int kk = 0; kk < wSamp / 2; kk++)
152.          {
153.              Y[kk][ii] /= fftMax;
154.          }
155.      }
156.  });
```

Appendix 10: MainWindow Parallel Code Changes

```
313.    int N = stftRep.timeFreqData[0].Length;
314.                Parallel.For(0, NUM_THREADS_USED, iterator =>
315.                {
316.                    int chunk_size = (N + (NUM_THREADS_USED - 1)) / NUM_THREADS_USED;
317.                    int start = chunk_size * iterator;
318.                    int end = Math.Min(start + chunk_size, N);
319.
320.                    for (int jj = start; jj < end; jj++)
321.                    {
322.                        for (int ii = 0; ii < stftRep.wSamp / 2; ii++)
323.                        {
324.                            HFC[jj] = HFC[jj] + (float)Math.Pow((double)stftRep.timeFre
    qData[ii][jj] * ii, 2);
325.                        }
326.                    }
327.                });
```

```
373.  Parallel.For(0, NUM_THREADS_USED, iterator =>
374.  {
375.      int guardSize = lengths.Count;
376.      int chunkSize = (guardSize + (NUM_THREADS_USED - 1)) / NUM_THREADS_USED;
377.      int start = chunkSize * iterator;
378.      int end = Math.Min(start + chunkSize, guardSize);
379.
380.      for (int mm = start; mm < end; mm++)
381.      {
382.          Complex[] twiddles;
383.          Complex[] compX;
384.          Complex[] Y;
385.          double[] absY;
386.
387.          int nearest = (int)Math.Pow(2, Math.Ceiling(Math.Log(lengths[mm], 2)));
388.          twiddles = new Complex[nearest];
389.
390.          for (int ll = 0; ll < nearest; ll++)
391.          {
392.              double a = 2 * pi * ll / (double)nearest;
393.              twiddles[ll] = Complex.Pow(Complex.Exp(-i), (float)a);
394.          }
395.
396.          compX = new Complex[nearest];
397.          for (int kk = 0; kk < nearest; kk++)
398.          {
399.              if (kk < lengths[mm] && (noteStarts[mm] + kk) < waveIn.wave.Length)
400.              {
401.                  compX[kk] = waveIn.wave[noteStarts[mm] + kk];
402.              }
403.              else
404.              {
405.                  compX[kk] = Complex.Zero;
406.              }
407.          }
408.
409.          Y = new Complex[nearest];
410.
411.          Y = fft(compX, nearest, twiddles);
412.
```

```
413.            double maximum = 0;
414.            int maxInd = 0;
415.
416.            absY = new double[nearest];
417.
418.            for (int jj = 0; jj < Y.Length; jj++)
419.            {
420.                absY[jj] = Y[jj].Magnitude;
421.                if (absY[jj] > maximum)
422.                {
423.                    maximum = absY[jj];
424.                    maxInd = jj;
425.                }
426.            }
427.
428.            for (int div = 6; div > 1; div--)
429.            {
430.
431.                if (maxInd > nearest / 2)
432.                {
433.                    if (absY[(int)Math.Floor((double)(nearest - maxInd) / div)] / absY[
    (maxInd)] > 0.10)
434.                    {
435.                        maxInd = (nearest - maxInd) / div;
436.                    }
437.                }
438.                else
439.                {
440.                    if (absY[(int)Math.Floor((double)maxInd / div)] / absY[(maxInd)] >
    0.10)
441.                    {
442.                        maxInd = maxInd / div;
443.                    }
444.                }
445.            }
446.
447.            if (maxInd > nearest / 2)
448.            {
449.                pitches[mm] = (nearest - maxInd) * waveIn.SampleRate / nearest;
450.            }
451.            else
452.            {
453.                pitches[mm] = maxInd * waveIn.SampleRate / nearest;
454.            }
455.        }
456.    });
```

```
749.    private Complex[] fft(Complex[] x, int L,  Complex[] fft_twiddles)
750.            {
751.                int ii = 0;
752.                int kk = 0;
753.                int N = x.Length;
754.
755.                Complex[] Y = new Complex[N];
756.
757.                if (N == 1)
758.                {
759.                    Y[0] = x[0];
760.                }
761.                else
762.                {
763.
764.                    Complex[] E = new Complex[N / 2];
```

```
765.                    Complex[] O = new Complex[N / 2];
766.                    Complex[] even = new Complex[N / 2];
767.                    Complex[] odd = new Complex[N / 2];
768.
769.                    for (ii = 0; ii < N; ii++)
770.                    {
771.
772.                        if (ii % 2 == 0)
773.                        {
774.                            even[ii / 2] = x[ii];
775.                        }
776.                        if (ii % 2 == 1)
777.                        {
778.                            odd[(ii - 1) / 2] = x[ii];
779.                        }
780.                    }
781.
782.                    E = fft(even, L, fft_twiddles);
783.                    O = fft(odd, L, fft_twiddles);
784.
785.                    for (kk = 0; kk < N; kk++)
786.                    {
787.                        Y[kk] = E[(kk % (N / 2))] + O[(kk % (N / 2))] * fft_twiddles[kk
    * (L / N)];
788.                    }
789.                }
790.
791.                return Y;
792.            }
```