Parallelization of Digital Music Analysis

HIGH PERFORMANCE AND PARALLEL COMPUTING

Sophia Politylo
N10489045 | CAB401

Table of Contents

1.0 Preliminary Research

1.1 Original function of the application Digital Music Analysis

The original Digital Music Analysis program was created to allow a user to compare the expected note from a music sheet, against a recording of music. For this to work however only a single instrument can be present within the recording. This is preformed through loading in a .wav file containing the music played to be analysed, along with a .xml that is a representation of sheet music. Given these two files the program is then able to analyses the .wav file pulling out the time, length and pitch of each note played on the instrument, the program than reads the sheet music and gets the relative time pitch and length for when each note should be played. With both the data for the notes played and expected notes, the program then iterates through both arrays comparing the pitches and length of the played note and expected note, with it then returning the relative error for each note.

After analysing the sheet music and played music the program will then output a GUI displaying three separate tabs. The first one is a histogram that just displays the frequencies of the sounds in the .wav file. The second tab shows the different note frequencies played through the song at a given time. The third shows a breakdown of the notes played with it showing correct notes in black and wrong notes in red along with the error and actual pitch played. Overall the created application when being paralysed will need to be able to successfully preform the functions described above, analysing the notes and frequencies, along with generating a GUI to display the relevant information.

1.2 Original Software Architecture Structure

The original application is based off C# using a windows form App, to enable a basic UI for the user to interact with. The created project has 7 .cs, files with 2 of them being tied to xaml GUI files. Out of these 7 files there is only 5 potentially relevant files for parallelisation, these are the “MainWindows.xaml.cs”, “musicnote.cs”, “noteGraph.cs”, “timefreq.cs” and “wavefile.cs”. the other two file “App.xaml.cs” and “AssemblyInfo.cs” are not important when it comes to parallelization given that these two files just provide information about the creation and functioning of the GUI. Along with the .cs files there is two xaml files “App.xaml” and “MainWindow.xaml”, both files are tag based files that are used to construct the forms for the GUI.

The entry point for the application is the MainWindow.xaml.cs file with it initialising all classes and variables needed, with it prompting the user to load in a .wav and .xml file for analyses. From there the MainWindow file has multiple functions it calls that generate objects based off the selected .wav and .xml file sending the results to the GUI for the user to view. The other 4 .cs files are all self-contained classes that allow the MainWindow to analyses and store the information about the music in organised self-contained elements, below is a breakdown of each class:

Wavefile

This class takes a filestream and constructs public data about the inputted wave file. This class has a wide range of variables that will be saved, but the important data is the wave and samplerate objects, given that these variables are accessed by functions outside this class. This class however does not provide too much details given the class functions are handled by c# libraries with most variables loading in the bit stream of the wave file. After that the class will than construct a wave file based off the input file stream through looping through the data and removing ay offset.

musicNote

This class is an object that stores and calculates the note type and relative error a given frequency is to that note. This class will first calculate the relative error and then use a switch stamen to determine the note.

noteGraph

This class is used within the GUI section of the program with it forming the backbone of the staff view, given that this class stores the positions of all notes, for drawing.

Timefreq

This class is used to load in the given music files bit stream, converting the input into it's time and frequency domain for future analyses.
This class stores three variables:

timeFeqData: float[][], containing the frequency of the music within the time domain.
wSamp: int, rate that the .wav file plays at or number of samples taken per second.
Twiddles: complex[] this is an array that stores the relevant transforming information to construct each point of the timeFeqData from the inputted wav file, through a Fourier transformation.

This function does not contain any functions that can be called outside of the initialisation of this class.
To generate the variables within this class the wave file stream has to be feed to it in a float array called x, along with the sample rate as an int variable called windowSamp.
Given these variables wSamp will become windowSamp. From here the class then constructs the Twiddles variable through iteration through iterating from 0 to wSamp, with it inserting a Complex number into the Twiddle array using the following equation. Where ii is the loop iteration I is an imaginary number.

```
double a = 2 * pi * ii / (double)wSamp;
twiddles[ii] = Complex.Pow(Complex.Exp(-i), (float)a);
```

With Twiddle constructed the class then sanitises the input file transforming the size of the input array into a multiple of the sample size, via inserting zeros to the end of the stream if the array stops before a multiple of the sample size. After setup the class than loops through the given sound file, preforming the function stft to each bit in the sound file.

Stft

This function takes a complex array and an integer for the sample rate and returns a 2-dimensional float array. The function stft works through setting up the time frequency data variables and calling the fft function to get the frequency data for the input wave file. To do this the STFT must first construct a clean frequency array, which than will have the finished data insert into. To get the frequencies from there stft will than iterate through the input wave file with it iterating through the sample rate of the wave file, calling the function fft for each sample in the wave file.

Ftt

fft function takes a complex array as the variable x and returns a complex array. This function forms the major bulk of processing time for this function with it being a recursive function that keeps calling itself twice every iteration until the complex array x reaches a length of 1. The last recursive function will return itself, allowing the other ftt's to perform a for loop to reconstruct a complex array, through adding up each part of the original input array with the corresponding multiplication from twiddles for a given iteration.

MainWindow

This class is a child of the pre-built window class in C#, because of this not all aspects of the functions and code base for this class can be modified, given that some of the functions are in built into C#. This class also servers as the main entry point into the application, with it loading in the files and calling all the other classes and functions. The relevant functions for parallelization within this class are the following:

loadHistogram

This function loads in the histogram on to the GUI and uses 4 for loops to place the octaves on to the graph area for the histogram

freqDomain

This function transforms the input wave file into the frequency time domain, calling the class Timefreq, with it also constructing a one-dimensional array containing the timefrequency data returned.

onsetDetection

This function uses the time frequency data to calculate the duration, start time and finish time of each note. This is done through looping through the time frequency data and selecting out the change in frequencies which are then added to two lists a stop and start. These points are than used to loop through the input wave file again having another transformation applied to it to get the frequency domain for each note, through selecting out the portion of the file that corresponds to the start and stop time of each note. To do this a fft function is again used within a loop, with it returning data that is than used to figure out the corresponding note for each frequency, this fft function however has a constantly changing twiddles variable given that the length of each note changes. This function also prints the staff on to the GUI.

Fft

This function takes three variables a complex array corresponding to the input wave file, the length of the file to sample. This function like the last fft function in the class timefreq is a recursive function calling itself

twice every iteration, until it reaches an array length of 1. At which point it will then preform the multiplication and additive function to the array, transforming the input into its time frequency domain.

readXML

this function takes a string which representants a xml file to open this function uses the C# class of filestream and XmlTextReader, to load in the input stream. Using these two classes nestled while loops are used to read the xml file and calculate the notes on the sheet music, with it than returning an array of the musicNote class.

stringMatch

this class takes in either 2 arrays containing integers or string, which represent the pitches to compare. this is done through looping through the input arrays comparing the surrodning pitches to one another to calculate two arrays representing the alignment of both the sheet music and music played.

1.2.1 Program Call Tree

To help visualises the call path of this program a call tree has been provided within figure 1, showing the flow and calls made. Using this call tree a few observations can be made about how the program calls each function. With the Main function call the ctor function which interns runs the MainWindow constructor function.

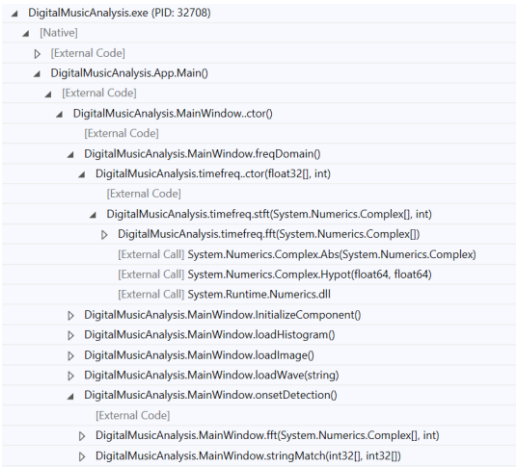


Figure 1 - main call tree for all function called on the intialisation of MainWindow

From there each of the created function can then be called with the function freqDomian initialising the class timefreq, which than calls timefreq constructor, calling the stft function which than calls fft, fft than recursively calls itself recursive 12 times as seen in figure 2.

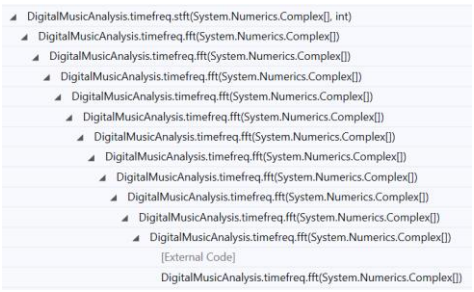


Figure 2 - call tree for stft and the recursive calls for fft

OnsetDetection is also called from the initialising mainwindow function with it in turn calling its own fft function within the mainWindow class. With fft recursively calling itself 18 times, as seen in figure 3. Apart from calling fft, Onesetdetection also calls the funcrion stringMatch after all fft have completed.

```
DigitalMusicAnalysis.MainWindow.onsetDetection()
 [External Code]
 DigitalMusicAnalysis.MainWindow.fft(System.Numerics.Complex[], int)
  DigitalMusicAnalysis.MainWindow.fft(System.Numerics.Complex[], int)
   DigitalMusicAnalysis.MainWindow.fft(System.Numerics.Complex[], int)
    DigitalMusicAnalysis.MainWindow.fft(System.Numerics.Complex[], int)
     DigitalMusicAnalysis.MainWindow.fft(System.Numerics.Complex[], int)
      DigitalMusicAnalysis.MainWindow.fft(System.Numerics.Complex[], int)
       DigitalMusicAnalysis.MainWindow.fft(System.Numerics.Complex[], int)
        DigitalMusicAnalysis.MainWindow.fft(System.Numerics.Complex[], int)
         DigitalMusicAnalysis.MainWindow.fft(System.Numerics.Complex[], int)
          DigitalMusicAnalysis.MainWindow.fft(System.Numerics.Complex[], int)
           DigitalMusicAnalysis.MainWindow.fft(System.Numerics.Complex[], int)
            DigitalMusicAnalysis.MainWindow.fft(System.Numerics.Complex[], int)
             DigitalMusicAnalysis.MainWindow.fft(System.Numerics.Complex[], int)
              DigitalMusicAnalysis.MainWindow.fft(System.Numerics.Complex[], int)
               DigitalMusicAnalysis.MainWindow.fft(System.Numerics.Complex[], int)
                DigitalMusicAnalysis.MainWindow.fft(System.Numerics.Complex[], int)
                 DigitalMusicAnalysis.MainWindow.fft(System.Numerics.Complex[], int)
                  DigitalMusicAnalysis.MainWindow.fft(System.Numerics.Complex[], int)
```

Figure 3 -call tree for onsetdetection and the recursive call for fft

The other functions called by the initialisation of the class MainWindow do not have any complex call trees, with the other functions not calling on any function or class created for this application, with them either calling on existing C# libraries or functions. The other functions are readXML, playback, loadFile, loadWave, LoadImage, loadHistogram and InitializeComponent.

1.2.2 Program dependencies

Just as important as the call tree is the dependences within the application, given that some functions and variables have data dependences making the order of execution of functions and classes very important for the integrity of the application. Bellow in figure 4, shows the original layout of the functions called within MainWindow's initialisation function. From this figure a few dependencies can quickly be seen with a few variables and classes. The first of these dependencies is with the object filename and the function loadWave() function, given that filename is an argument for this function. Another dependency is the variable xmlfile and the object sheetmusic given that the object shettmusic is an object of class readXML which takes as its argument xmlfile.

```
  InitializeComponent();
filename = openFile("Select Audio (wav) file");
string xmlfile = openFile("Select Score (xml) file");
Thread check = new Thread(new ThreadStart(updateSlider));
loadWave(filename);
freqDomain();
sheetmusic = readXML(xmlfile);
onsetDetection();
loadImage();
loadHistogram();
playBack();
check.Start();
```

Figure 4 - MianWindow initialisation code showing the order of the functions called by the program

With the information known about the classes other code dependency can be found with freqDomain needing the function loadWave to be completed before executing, given that the class timefreq requires the wave file's data stream and sampling frequency. onsetDetection requires both the loaddWave(), freqDomain() and sheetmusic object to be constructed and finished. Since onsetDetection() uses the timefrequency domain to get the start and stop time of the notes, with it than constructing the pitches of the played notes using the wave file's data. Along with this onsetdetection than compares the pitches against the xml file loaded into the object sheetmusic. loadImage() only requires the function freqDomain to be completed given it uses variables from the class stftRep which is constructed in the function freqDomain. loadHistogram() requers onsetDetection to be completed given that it needs the notes and relative errors to produce the sheet music view. playBack() only requires the function loadwave to be run, this however might be slightly miss leading given that this starts the playback of the wave file and is best done at initialisation of the GUI after all the other functions. The other function updateslider has only one dependency which is within the function PlayBack() given that this is a GUI component that shows the current location of the wave file's playback position. Along with this all functions using a GUI variable cannot be executed before InitializeComponent has finished, these functions are onsetDetection, loadImage, LoadHistogram and updateSlider. With this information the flow dependency diagram shown in figure 5 was created, showing the dependency of each function to the other. The arrow on each connection points to the function or object that the above function is dependent on.

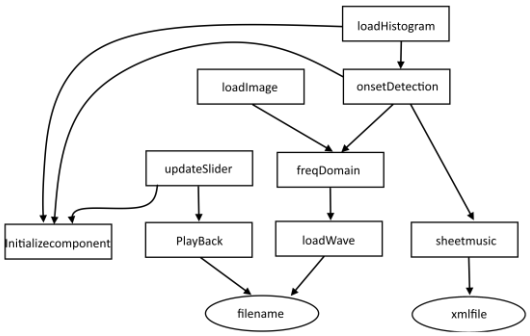


Figure 5 – function dependency graph for all functions within MainWindow's initialisation function

1.3 Analysis of Potential Parallelism within the Software

To successfully parallelise the software to increase speed potential code optimisations and loops will need to be found. To do this the first step is to find the major bottle neck functions within the code and analyses this function for potential parallelism and data optimisation. To ensure that no human error was present within the time calculation a few alterations were made to the code to remove any user input. These lines of code were lines 34, 35 and an added line after line 45 in the MainWindow.xaml.cs file with the new initialisation function for MainWindow in figure 6. With the filename and xmlfile both being changed to static fields, so that slection time would not be counted in total run time and CPU utilization. Along with the added line underneath check.start(), given that this will close the application after all the functions have run given a constant stop time for the CPU and time analyses.

```
        public MainWindow()
        {
            InitializeComponent();
            filename = "C:/Users/polit/Desktop/Digi/music/Jupiter.wav
            string xmlfile = "C:/Users/polit/Desktop/Digi/music/Jupiter.xml
            Thread check = new Thread(new ThreadStart(updateSlider));
            loadWave(filename);
            freqDomain();
            sheetmusic = readXML(xmlfile);
            onsetDetection();
            loadImage();
            loadHistogram();
            playBack();
            check.Start();
            System.Environment.Exit(1);
            button1.Click += zoomIN;
            button2.Click += zoomOUT;
            slider1.ValueChanged += updateHistogram;
            playback.PlaybackStopped += closeMusic;
        }
```

Figure 6 - New code in MainWindow's intialisation function to remove user input and the GUI for profiler analyses of the software

With this modified code DotTrace was used on the release build of the digital music analyser application to profile the code, producing the results seen within figure 7, With a total time of 10.354 seconds. When looking at the hotspots for the application two major bottlenecks are apparent with the fft in the class timefreq and fft in MainWindow, taking up the large amount of time.

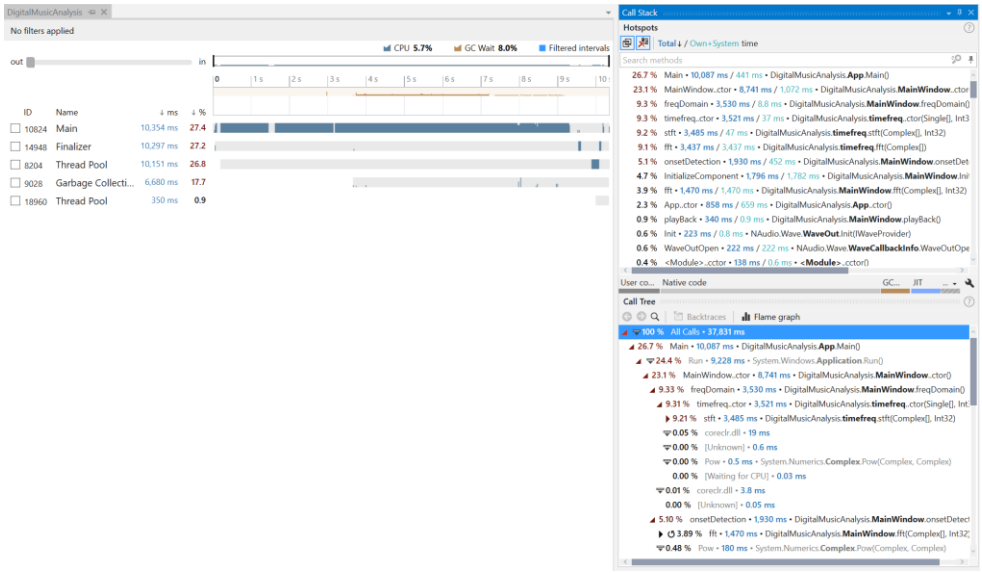


Figure 7 - DotTrace's performance analysist for the original sequential application

Using the time taken for both fft, their parent functions and the initialiseComponent function table 1 was produced, which shows the time and the total percentage of time each function took when compared to the whole program. From these standardised times it can be seen that the fft functions are major bottle neck for their parent functions stft and onsetdetection, with stft only taking 0.46% of the total time when removing fft from it, while Onsetdetection takes 6.9% of the total after removing fft. The initialiseComponent time is also a fairly large bottleneck taking 17% of the total time, this function however cannot be parallelised, with it instead being able to run in the background of the fft calculations.

Run Time						
Function	**Main**	**`stft**	**fft**	**ondect**	**fft_main**	**init**
time	10.354	3.485	3.437	1.93	1.214	1.796
standardised time	100%	33.6585%	33.1949%	18.6401%	11.7249%	17.346%

Table 1 - time taken by the fft functions and their parent functions along with the percentage of time each function took

Using the standardised times Amdahl's Law can be used to predict the theoretical maximum scalable speedup of the application. To get the percentage that can be parallelised the fft functions will be added together to get the max scalable parallelisation. Doing this gives a total percentage of 45% of the application with scalable parallelism. Using this percentage, the theoretical speedup graph in figure 8 was created, with a max possible speed up being between 1.7 and 1.8, with it plateauing just before 1.8.

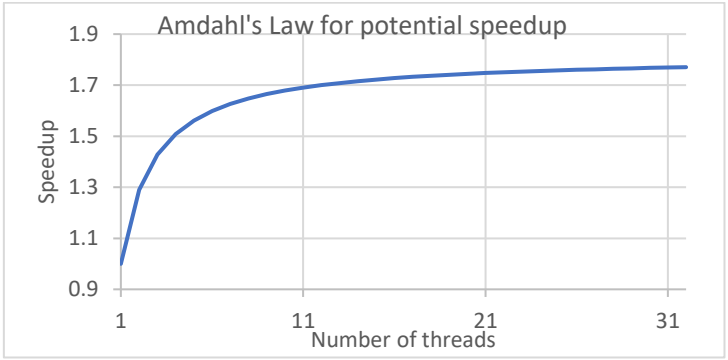


Figure 8 - theoretical speedup that can be achieved by the application using Amdahl's Law

When looking at the calls to fft both stft and fft use this function. The important code that calls fft in stft can be seen within figure 9. From the code that calls fft, one optimisation could be made for parallelisation this is in the form of turning the outer ii loop into a parallel function given that there are no data dependences within the ii loop. Doing so would enable interactions of ii to run in parallel, thus making the fft function also run in parallel to itself cutting down on computation time. There is however single problem with the fftMax check at the bottom of the code, given that each thread will need to access fftMax to check against and update the value. This howerver will create a race condition if a process reads the fftMax variable just before another process updates the value, as this would cause the other thread to be checking against a lower fftMax than the current max causing the thread to possibly insert a lower value than the true max value. To fix this a lock could be created surrounding this check to ensure that no process is accessing fftMax at the same time.

```
for (ii = 0; ii < 2 * Math.Floor((double)N / (double)wSamp) - 1; ii++)
          {
              for (jj = 0; jj < wSamp; jj++)
              {
                  temp[jj] = x[ii * (wSamp / 2) + jj];
              }
              tempFFT = fft(temp);
              for (kk = 0; kk < wSamp / 2; kk++)
              {
                  Y[kk][ii] = (float)Complex.Abs(tempFFT[kk]);
                  if (Y[kk][ii] > fftMax)
                  {
                      fftMax = Y[kk][ii];
                  }
              }
          }
```

Figure 9 – code for stft's for loop that calls fft

A lock however is no prefect as it would mean each process will need to wait to access fftMax, slowing down the total run time. A solution to this could be to parallelize the fft code directly. Figure 10, show the code for the fft function. Within this code section a few potential fixes could be attempted the first is to parallelize the for loop at the end, this however comes with a few problems. The first one being that a check for the array size would need to be created given that N could be as small as 2, making parallelisation slower than a sequential solution, given the overhead in thread creation would be greater than the computationsdone. The other downside for this approach is due fft being recessive with each fft that has an N greater than 1 producing 2 more fft functions. Due to this fact parallelizing fft has the potential to produce to many process that cannot terminate, since fft recursively calls itself meaning that each time a thread is create two more will be created, with child process waiting for a free thread. Which once all parents have saturated all the threads cannot ever become free due to the parent not terminating until all children have been returned.This bottleneck condition also stops the other potential parallelisation that could be done to the fft function, such as parallelisation through calling both the fft odd and fft even at the same time in parallel. Since again the same bottleneck will occur with parent functions using up all the threads while waiting for children process that are waiting for free.

```
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];
            }
        }
```

Figure 10 - code within the fft function

Because of this deadlock that occurs from thread saturation that recursive functions because it is impractical to parallelize fft as is. Due to this the best course of action for fft would be to reformat the code so that recession can be eliminated so that no process is waiting on any child process to finish allowing for CPU resource to be freed up and reallocated when needed.

When looking at the second fft function within the class MainWindow, the same parallelization techniques can be applied with the original fft caller being the function onsetDetection with the code for this loop in figure 11. Again, the outer loop could be parallelized but again a problem occurs in how the note pitches are stored in a list, since insertion of variables will become unstructured when parallelized given that each process will be accessing the list at the same time and list can only store new objects at the bottom of the stack with no why to dynamic insert objects. To fix this problem the pitch list could be transformed into an array with the size of the lengths list.

```
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){
        pitches.Add((nearest - maxInd) * waveIn.SampleRate / nearest);
    }
    else{
        pitches.Add(maxInd * waveIn.SampleRate / nearest);
    }
}
```

Figure 11 - onsetdetction function's loop that calls fft code

This again leaves the second option for fft to be analysed which again produces the same results as last time given that again fft is a recessive calling itself two times if N is larger than 1. Which again means that the only way to parallelize the MainWindow fft function directly is through refactoring the code.

With scalable parallelism within the fft function being exposed, the other major bottleneck InitializeComponent can be possible parallelised to increase speedup. However, given that this function is a generic initialiser for the C# class Window, no scalable parallelism can be created through refactoring the InitializeComponent function. However, InitializeComponent can be parallelised to run in the background of the timefreq, loadWave and sheetmusic function, given that these functions have no GUI elements. Onsetdection can also potentially be parallelised to run while InitializeComponent is running given that the GUI elements of this function are separate to the pitches and frequency calculations. The theorised parallelisation of functions that could than be created from this method are shown in figure 12, with the onsetdetection GUI functions being broken off into the new function staffDisplay. The flow of the functions is shown with the arrows, with the head pointing to the function that is dependent on the last. For loadImage and staffDisplay two locks will have to be made that will stall the Main thread until freqDomian is finished before moving on to loadImage and a second lock that ensures that staffDisplay is only executed once onsetdetection has been completed. Employing this parallelisation technique should remove at best 17% of the total run time for the program given that InitializeComponent took 1.796 given that it both the music note calculations and window screen setup can be competed at the same time. Doing this however will make it harder to see scalable parallelism within the fft function given that after freqDomian and onsetdetection take up less than 17% of the run time any further improvement is obscured by the InitializeComponent function.

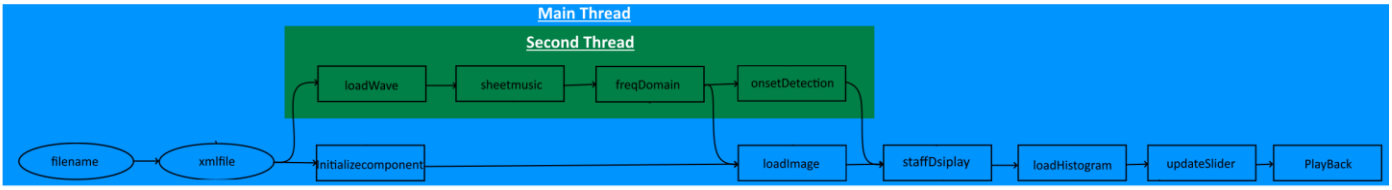


Figure 12 - parallel thread structure for the MainWindow's initialisation function

2.0 Parallelization of Application

2.1 Mapping Computational data and processors

To create the parallel application the CPU was targeted using the .NET Core Threading library, with the classes threadpool and Thread class being used when necessary to create the separate parallel process. The .NET Core threading library was chosen since it provides a strong base for any threading application that utilises CPU threads for multitasking. Using the CPU was important given that this application's main target is home devices, which might not have a GPU to utilise.

The two main classes used are the threadpool class and Thread class. Both classes provided a unique and useful parallel application of threads. The threadpool class is incredibly useful for recycling threads, removing the amount of garbage collection and thread creation each new parallel task needs. Due to this fact threadpool is an important class for parallelizing loops, since it removes the need for creating new threads for each for loop, and thus lowers the overhead of parallelizing the for loop. Which is very important when multiple loops are being parallelised in an application with the thread only needing to be created once and destroyed when the software ends. There is however one drawback to a threadpool which is that there is a max number of threads than can be utilised using this method with that usually corresponding the number of threads the machine possess. This is not by itself a drawback given that it stops the creation of too many threads.

The Thread Class however provides more direct control over the creation of threads with each thread being created at the specified point in the code and then getting destroyed at the end of the given task. This makes the Thread class much more suited for long term parallel part of code given that there is a construction and destruction overhead for these threads making them impractical for loop parallelization, but instead a much better candidate for making function or user input run in parallel with other functions.

2.2 Code added/Changed to Achieve Parallelization

To expose parallelism within the program a few major functions and parts of the code had to be refactored to remove data dependencies to archive parallelization or better caching through restructuring the data.

2.2.1 Parallelizing MainWindow's Initialization function

The first action undertaken was to separate the GUI components from the freqtime and pitch calculation of onsetdetection, so that InitialiseComponenet could run in parallel. To do this nothing special was done to timefreq with its only dependency being the loading in of the wave file. To enable InitialiseComponenet and onsetdetection to run in parallel, the GUI rendering component needed to be decoupled from the note detection. This is a simple enough change given that note detection is performed at the start of the function; in its own loop, with onsetdetection only rendering the staff components after all notes and note offsets have been calculated. This means that onsetdetection can be easily split into two new functions. The original onsetdetection function which gets the timing and pitch of the notes and a second function called staffDisplay (lines 543 - 679), which renders the notes onto the GUI. To achieve these two variables that were originally only local to onsetdetection were made global so that the new function staffDisplay could also access these variables.

After this a second thread was created within the initialisation of MainWindow with the functions loadWave,freqDomain, sheetmusic and onsetdection being performed in the second thread (lines 37 - 54). The second thread will run in parallel to the main thread executing. THE GUI elements InitialiseComponenet, loadImage, loadHistogram and staffDisplay, with both loadImage and LoadHistogram needing timefreq to initializes the object stftRep before running. To ensure that this condition is meet a lock was created to check if timefreq has been completed, with the main thread being locked after InitialiseComponenet if timefreq has not been finished.

2.2.2 Parallelizing sftf

To parallelise the stft loop fftMax had to be removed from the for loop calling fft, given that in parallel fftMax has the potential to not contain the largest value returned by an fft function, given that the parallel functions have a high likelihood of accessing fftMax while other thread are causing a collision. To fix this problem each thread will calculate a local fftMax variable, from their computed fft functions, which will be saved in a global float array called fftMax_array that is the length of the number of threads (lines 105 - 136). Once all threads have completed and returned there local fftMax to the global array, the main thread will than check each of the local fftMax and set the fftMax as the highest local fftMax in the array.

2.2.3 Parallelizing onsetdetection

To expose parallelism within the onsetdetection loop that calls the fft function the order of the notes returned to the list pitches by the fft function need to be kept. But given that the current storage method is a list, and the access pattern of the threads are unknown this order will not be retained. To fix this problem the list can be transformed into an array with the length being set to the length of the array lengths, given that is the number of times that fft saves a value to pitches. With this array each thread can now be given a relative range to save the fft result to given that each thread preforms a slice of the original mm loop which will be iterated upon for the length of lengths, meaning that the mm slice used by for each thread is also the slice of the pitches array to save to (lines 370 - 486). Onsetdetection has one more problematic data dependency preventing parallelisation, which is the twiddle variable fft uses. Given that it is originally a global variable that is updated just before calling fft in the onsetdetection's mm loop. Which in turns means that if done in parallel the twiddle variable will be constantly overwritten by the various threads since every thread access the same twiddles. To overcome this obstacle twiddles will be transformed into a local variable for each thread that is passed into the fft function. Via doing this it will ensure that each process can calculate their own twiddles value without effecting the other processes.

2.2.4 Improving the data access pattern for fft

The last change made in the code was made to help with the memory and data usage of the fft functions. This optimisation was important to ensure that better caching could be achieved by each process, increasing the speed of the program through removing time spent on initialising and fetching variables from memory. Given that fft is a recursive function that generates the equivalent of the original array $\frac{\ln N}{\ln 2}$ times with each parent array combining the two child arrays into a larger array. Due to this the caching ability of this function is weak given that each iteration of fft requires a new array to be created and cached. To fix this problem the code was refracted so that only two or three arrays were needed along with a few local variables to calculate the final fft array.

Given that fft is recursive with each array being broken up into even and odd locations and sent off to two separate fft functions, with the returned odd and even variables than being returned and operated on in the following for loop seen in figure 13. N in this code is the size of the array which recursively gets smaller until it reaches 2. It will than add the even and odd components in the same location with it multiplying the odd component by a variable in the twiddles array. The twiddles variable however is dependent on the current kkk iteration and given that kkk will go to double the length of the odd and even array, once kk reaches N/2 it loops back to the first element in the array this means that the twiddles for the first occurrence of each even and odd value will be different to the second occurrence.

```
for (int kkk = 0; kkk < N; kkk++){
                Y[kkk] = E[(kkk % (N / 2))] + O[(kkk % (N / 2))] * twiddles[kkk * wSamp / N];
            }
```

Figure 13 - operation preformed at the end of each iteration of fft

Due to this transforming the recursion function into a single for loop is not as simple. However as seen in figure 14, which steps out the process happenes with the kkk loop and the twiddle indexes as each fft returns to its parent, a pattern with how the elements are added and how the twiddle multiplier is calculated can be seen. From this it can be assumed that each iteration of the main loop splits the array by half, so for iteration 1 the array is 1 and the twiddles would have an index multiplier of 0 or 1 given that is the number of arrays, from there the number of arrays will be multiplied by 2 to get the new number of arrays. Which now means that a second loop will need to be used to get the elements from 0 – N/2 and a second for N/2 – N. The twiddle multiplier this time however will be the last twiddle multiplier plus the length of the sub arrays divided by 2 for odd index in each sub array and the last twiddle multiplier for even numbers.

Array																length	gap	# arrays
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16			
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	16	8	1
0	0	0	0	2	2	2	2	1	1	1	1	3	3	3	3	8	4	2
0	0	4	4	2	2	6	6	1	1	5	5	3	3	7	7	4	2	4
0	8	4	12	2	10	6	14	1	9	5	13	3	11	7	15	2	1	8

Figure 14 - step by step view of the changing twiddle index for each iteration of fft for an array with the max size of 16 elements

Given this basic loop a fft can be transformed into three for loops that are able to access the original array to perform the additive function for each element in the array, the following sudo code in figure 15 shows the required code structure (lines 139 – 186). This code produces the same outcome as the original fft function with less data overhead, significantly increasing the speed of fft with a speedup of 2.8 in tests. The sorting aspect of this function works as the last twiddle multiplier also corresponds to the location of each variable in the final x array, given that in the original multiplier and array location is calculated in a for loop.

```
Takes array of  x
Int N=length of x
aSize=N
gap=N/2
SubArrays=N/aSize
SubSize=2
mulArray=array of N length
For split=N/2 while split > 0
        For s=0 while s<SubArrays:
                For i=0 while i<gap
                        E=x[p+s*asize]
                        O=x[p+gap+s*asize]
                         mul1 = mulArray[p + s * aSize];
                         mul2 = mulArray[p + gap + s * aSize];
                        x[p+s*asize]=E+O*twiddles[mul1 * wSamp / subSize]
                        x[p+gap+s*asize]=E+O*twiddles[mul2 * wSamp / subSize]
                        I++
                S++
        Split=/2
        aSize/=2
        gap/=2
        SubSize*=2
        SubArrays=N/aSize
For i=0 while I < N/2
        Temp= x[mulArray[i]]
        x[mulArray[i]]= x[i]
        x[i]=temp
return x
```

2.3 Parallelization Results and Analysis

Before any analysis of the final software can take place, the results produced by the parallel and sequence software must be compared to ensure that the desired output and GUI are rendered correct. Given that the created application must correctly display the errors in the music played compared to the sheet music the most important results to test for are output frequency histogram and the staff screen with the relative error for each note. To test the accuracy of the created parallel application the wav file Jupiter.wav and Jupiter.xml file was used to produce the results seen within figures 15, 16, 17 and 18.

Figure 15 displays the frequency histogram for the parallel application while figure 16 shows the frequency of the sequential application. Both produced histograms when compared to each other produce the exact same results meaning that the parallelization does not affect the desired output of the histogram.

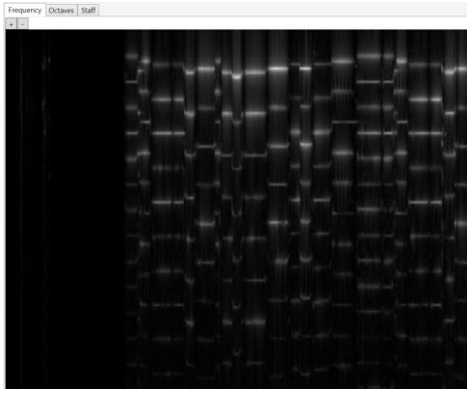


Figure 15 - histogram for parallel application

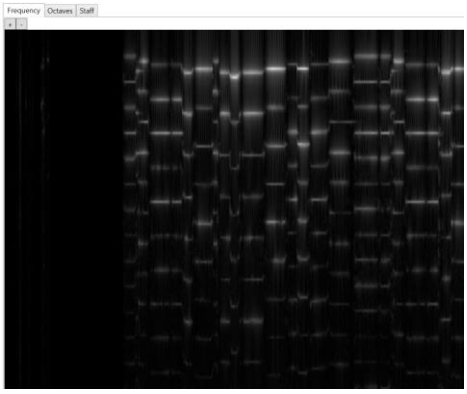


Figure 16 - histogram for sequential application

The more important check however is the staff view and the relative error for each note in the sheet music and the played note. Figure 17 shows the staff view for the parallel application, while figure 18 shows the staff view for the sequential application. From both these figures there is no errors introduced to the desired results with all error notes (in red) being produced at the same position while all correct notes (in black) are also produced in their correct position. There is however another check that can be performed on the staff screen and that is with the expected pitch, actual frequency, and pitch error labels. The note that is being looked at is shaded in yellow in the figure, with both notes producing the exact pitch of G_, same frequency of 783.9921745898773 and the same pitch error of 0.0028764993921055. Which again reenforces that parallel application does not change the desired result from the sequential application.

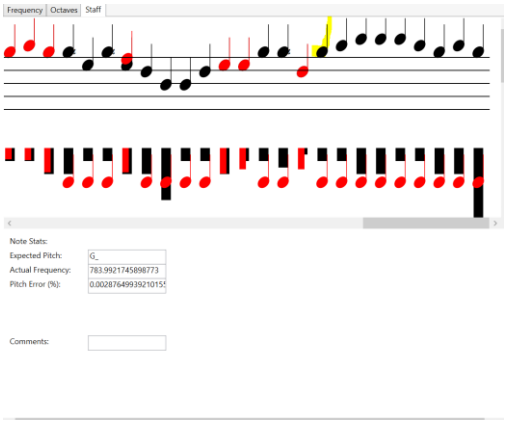


Figure 17 - staff view for parallel application

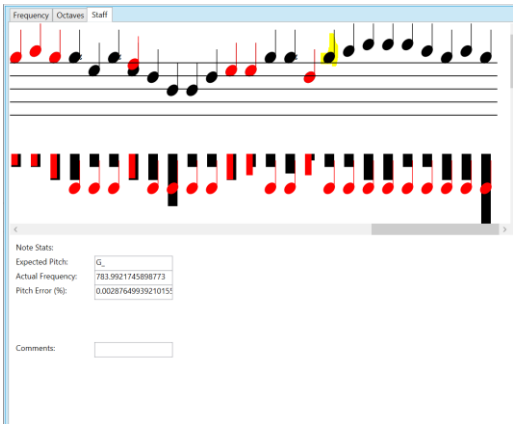


Figure 18 - staff view for the sequential application

To make sure that the program functions the same with a different wave file input a second test was done using the same .xml file, with the wave file being changed to B.B.scale.wav. This produced the flowing figures bellow for the staff view. With figure 19 being the parallel application while figure 20 is the sequential application. Again, both figures show the exact same output with all data being the same. With the same pitch of A_ being calculated, with an actual frequency of 880.0014621486731 and a pitch error of 0.00287649939139100 for the highlighted note in both images. These two test cases demonstrated that the created parallel application does still produce the expected output.

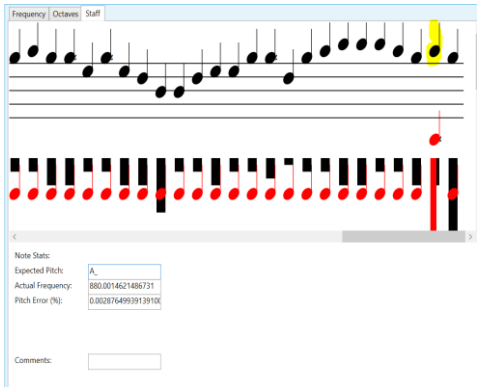


Figure 19 - staff view for parallel application using B.B.scale.wav

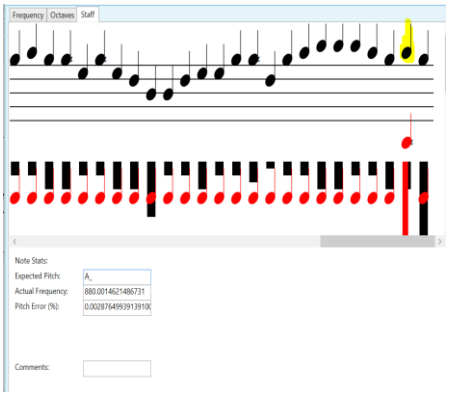


Figure 20 - staff view for sequential application using B.B.scale.wav

The final Parallelized software produced remarkable results when compared to the original software with it achieving a speedup of 4.05 for the fft function, and a speed up of 3.34 for the whole application. Figure 21 shows the speedup of the created application using a variety of cores from 1 to 32. This speed up results do produce a speed up much great than what was expected, but however do plateau much earlier than expected with speedup plateauing at around 8 threads, when Amdahl's Law predicted a plateau at around 11 threads. This however is most likely caused by the architecture of the non-scalable parallelisation preformed on the InitializeComponent and GUI functions which will always take 1.769 seconds to complete. Which in turn means that even if the fft functions completes before the GUI components the speedup will not be noticeable due to the application now needs to wait on InitializeComponent to finish. Meaning that the plateau will happen much earlier due to the maximum speedup being achieved when timefreq and onsetdetection take the same time as InitializeComponent. This however does not explain the decrease in speed that comes at 16 and 32 threads, both these values most likely are caused by thread creation overhead and there only being 16 threads. Which means that even at 16 threads a few threads will most likely be waiting for free resources due to the parallelisation of InitializeComponent using up a process and end background applications such as the operating system, using up a few threads. Due to this it means that the program is running on a fixed number of cores and if that limit is passed the program will operate at the same speed as the max threads available to the fft functions plus the overhead for creating the extra threads that need to wait.

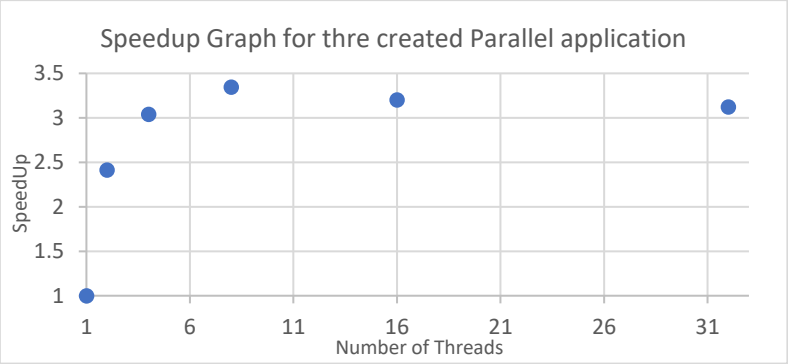


Figure 21 - Speedup Graph for the final created application showing the effect of increasing the number of threads for the stft and onsetdetection function

The new fft created function produced a speedup of 2.8289 when compared to the original fft function when utilising one thread. To prove that parallelizing the fft function in the stft loop does provide scalable parallelism the speedup graph in figure 22 was created. With the speed up plateauing at around four threads with a speed up of 4.036, this speed up however does slowly increase after 4 threads with it reaching 4.05082 at 32 threads. This scalable parallelism is less than optimal given that it plateaus early on minimal cores. This however could come from the total number of iterations being too small to see a noticeable difference from increasing the thread count past four, due to overhead.

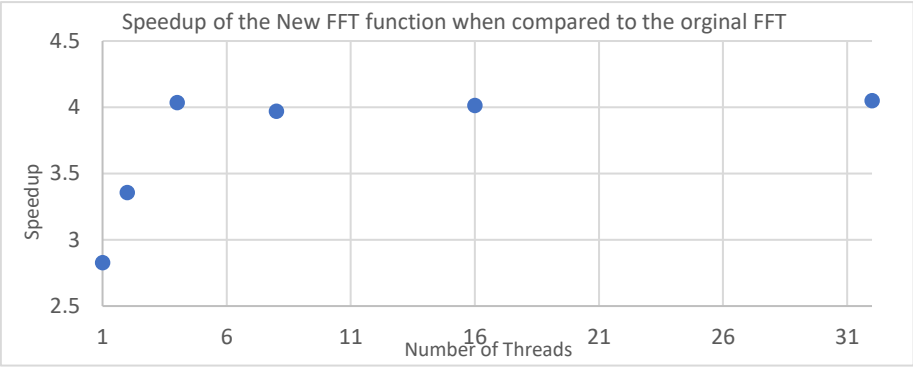


Figure 22 - speedup graph for the stft loop running in parallel using the new fft function

Overall, the created parallel program performs quite well when compared to the original with the fft functions providing scalable parallelism, while the non-scalable parallelism allows for the overall time of the program to reach the max speedup using less cores, given that it allows for the scalable parallelize to run parallel. Using these two methods have thus produced a total maximum speedup of 3.261102 or around 1.49 higher than the theoretical max speed up calculated from Amdahl's law.

2.5 Barriers preventing Parallelization

When it comes to parallelization there were a few barriers that proved challenging when trying to increase performance. These two major barriers come from the fft function. The first one of these barriers was within the onsetdetection fft, while the second one come about when trying to parallelise the loops within the fft function.

The first barrier is caused by the new fft function when running in onsetdetection, as an unknown error that kept getting thrown when trying to return the final array. This error was a System.IndexOutOdRangeException that is thrown when trying to return x. To try and fix this problem multiple fixes were attempted. The firs one of these was to check if the array that fft was returned had the right length. To check this fft was returned to no variable and another one was returned to a newly constructed complex array that had not yet been written to. Both of these tests however still returned the same error. This error could maybe have been thrown earlier in the function and only showed up at the return point. To check this every possible array index access variable was tested to ensure that they were within the range of 0 and the length of the array -1. These checks however still didn't catch any accesses outside the length of the array.

The last possible issue causing this error could be due to the length of the complex array being passed to fft. To test this theory the length of the array producing the error will be printed out and be used to set the max array size to pass to the new fft function. Doing so found that the max array size that can safely be passed to the new fft function is under 65536, with any array over this value producing the error. Knowing this the only fix to this issue is to use the new fft function for arrays smaller than 65536 and to switch back to the original fft function for arrays larger. Doing this still produces a remarkable speed up in the onsetdetection function with using just a single core producing a speedup of 1.88 when compared to the original code.

The second problem with parallelisation however could not be overcome unlike the first one, given that it was a problem regarding the speedup of fft using multiple threads. This problem was caused due to the speedup being worse when paralleling the loops within the fft function when compared to the stft and onsetdetection loops. This problem however is not big given that parallel speedup was still achieved just in a different manner than what was initially assumed to produce les performance. Figure __, shows the profiler performance of the parallelized fft loop while figure __ shows the performance of the parallelized stft loop. From these two figures parallelizing the stft loop that calls the fft function produces a speedup of __, while parallelizing the fft loop produces a speed up of only __. This should have been expected however given that in the stft function the fft loop has 2048 interactions while 2327 for the test wave file, which by itself is not a large discrepancy. However when analysing the loops, fft would be called 2327 times which means that each parallel thread for the fft loop would create and destroy the request amount of threads 2327 times, while paralleling the stft loop will only create and destroy the requested amount of threads, greatly reducing the overhead of creating threads along with providing each thread with much more information to work with making each thread overhead smaller when compared to the amount of work each thread will perform.

2.4 Tools and Software Used in Parallelization

The tools used to help parallelizes this software are Visual Studio 2019 and JetBrains dotTrace 2021.2.1. Visual Studio 2019 was both used as an IDE and a performance profiler for the created application. Visual Studio's performance profiler is a useful tool for parallelization given that it provided an easy and quick way to analyses and recoding the CPU usage of a program while in development. The VS profiler is also very handy for view the call tree for a given application since it allows for an easy to access view that allows for quick switching back and forth from the IDE code editing and profiler. The VS profiler however is not the best given that it provides limited information on how each thread is being utilised along with it only analysing the code created for a given project with it ignoring C# libraries and Classes.

JetBrains dotTrace 2021.2.1 is a more in-depth profiler that will not only show the overall CPU usage but break down the thread utilization to the thread level. This is very important given that it allows for problems within the parallelization to appear as now wait time for each thread can be seen and analysed allowing for a greater understanding on what should be moved and what can be performed while the main thread is waiting for a child to finish. Along with this dotTrace 2021.2.1 will also show when computation time is increased due to too many threads being utilised allowing for bottlenecks to become apparent when too many threads are being used.

3.0 Conclusion

The final created parallel program using the original sequential Digital Music Analysis program has surpassed the expected speedup that was originally expected, with the final program reaching a speed up of 3.3 for the parallel application at 8 threads. During this parallelisation process a range of lessons were learnt regarding the effectiveness of different strategies for parallelisation with it being found from attempted parallelisation that the outermost loop should always be parallelised, when compared to parallelising an inner loop. This is nothing when compared to what was learnt about how important proper data structuring is for speedup with refactoring fft producing a speedup of 2.8, through restructuring how the data was accessed and saved within the fft function. Due to these facts, the produced parallel program was very successful in producing a great speedup value for the Digital Music Analysis application, while also serving as an important learning experience. This application however could still be improved, with slightly more parallelism still being untapped in the final application, with a few more for loops being able to be potentially run in parallel. These loops however did provide enough potential speedup to be worth parallelising when compared to the loops that call fft and InitializeComponent, due to the other loops taking up such a small amount of total processing time. These loops however are worth another look if this application were to be further built upon but are most likely inconsequential with future improvement coming from further improvements on the caching and memory management of the existing functions.

5.0 Appendix

5.1 Deployment of Application

Within the assignment folder there has been included two versions of the application in the folders 'Original Digital Music Analysis' and 'Parallelized Digital Music Analysis'. With in both of these folders there is 2 sub directories with one being the compiled execution file for a final working release version for both projects, and a second folder called 'Visual Studio Project' that contains the Visual Studio project files for the application.

5.1.1 Original Application Location

Within the 'Original Digital Music Analysis' file is the base unedited code for the original application without parallelization. With the executable for the original file being in the sub directory 'Original Digital Music Analysis exe' with the .exe file 'DigitalMusicAnalysis.exe' being the complied project for the n parallelized application. The file 'Visual Studio project Original' contains the visual studio project for the original un parallelized application. With this project the user can edit line 45 in 'MainWindow.xaml.cs' via uncommenting the command there to set a hard end point for the application for testing. Along with this line 34 and 35 can be edited so that the two variables on those lines are pointing to the commented-out location if testing need to be done with automatic input. To run the new build the user can run it or build it like any Visual Studio application with the green arrow saying DigitalMusicAnalysis. This will run the program and create a new exe file at the location '' Original Digital Music Analysis\Visual Studio project Original\DigitalMusicAnalysis\bin" either in the Debug or Release folder depending on the chosen build type.

5.1.2 Parallelized Application Location

The parallelised application can be found within the second folder within the main directory with the file being called "Parallelized Digital Music Analysis". Within this fie again there is 2 folders one called "example 4 core build" which contains a build version for the edited parallelized release version of the application utilizing 4 threads, with the execution file being called "Digital_Music_Analysis_C4.exe". The Visual Studio file can be found within the folder called "Visual Studio project Parallelized", with this file the visual studio project used to create the application and execution file can be found. Within the project the user can edit line 53 in 'MainWindow.xaml.cs' via uncommenting the command there to set a hard end point for the application for testing. Along with this line 35 and 36 can be edited so that the two variables on those lines are pointing to the commented-out location if testing need to be done with automatic input. Along with this the user can edit the core_count variable at line 367 to any number of threads they want the parallelized loop to run across. The other file the user can edit within this project is 'timefreq.cs' with the user being able to edit the variable core_count at line 60, to enable the parallelized for loop to run across the desired number of cores. To run the new build the user can run it or build it like any Visual Studio application with the green arrow saying DigitalMusicAnalysis. This will run the program and create a new exe file at the location ''Parallelized Digital Music Analysis\Visual Studio project Parallelized\DigitalMusicAnalysis\bin" either in the Debug or Release folder depending on the chosen build type.

5.1.3 Running the application and GUI layout

After opening the application, the user will be prompted to open a .wav file, two different .wav files have been included in the music folder within the main folder, called Jupiter.wav and B.B.scale.wav, the user can either select one of these or their own .wav file.

After opening a .wav file the user will then be prompted to open a .xml file that contains the sheet music to read, one .xml file that is compatible with the software has also been include in the same file as the .exe called Jupiter.xml, which the user can load into the application.

From there the program will run and open the GUI screen and play back the .wav file to the user. The user can then use the tabs at the top to look at the different screens, with the fist screen the user sees being the histogram showing frequencies of the .wav file in an image format. The second screen shows the different pitches played at each interval within the .wav file. The third screen than shows the staff screen showing the sheet music, with the notes played. With red notes showing notes played incorrectly while black notes have been played correctly, the user can than mouse over each note to see the note pitch, note and the relative error between the expected frequency and played frequency.

5.2 Table of changed lines changed

5.2.1 MainWindow.xaml.cs

Function	Lines	Description
Stft	60 – 66	Adds variable definitions to enable parallelization of stft loop
stft	72 – 87	Adds the thread creation loop for the sft loop
Stft (par)	105 – 136	Function to run in each thread. Modified computation originally preformed in the stft loop
Fft	139 – 186	This is a modified fft function that removes any recursion from the function increasing the caching efficiency and memory management of the function

5.2.1 timefreq.cs

Function	Lines	Description
MainWindow	37 – 54	These changed lines added parallelize to the window constructor allowing for a more efficient execution of the various functions
onsetDetection	370 – 392	These changes were made to add parallelism within the onsetdetection, via spiting up the fft function into multiple threads for execution.
onsetDetection (par_mm)	393 – 486	This is a modified set of computations to be performed by the threads within the parallelized onsetdetection loop.
staffDisplay	543 – 679	This is a new added function with code being taken from onsetDetection, so that onsetdetection can run in parallel to the initialization of the GUI components. Through removing the GUI drawing onsetdetection into this new function.

5.3 Profiler screenshots

5.3.1 Whole program performance

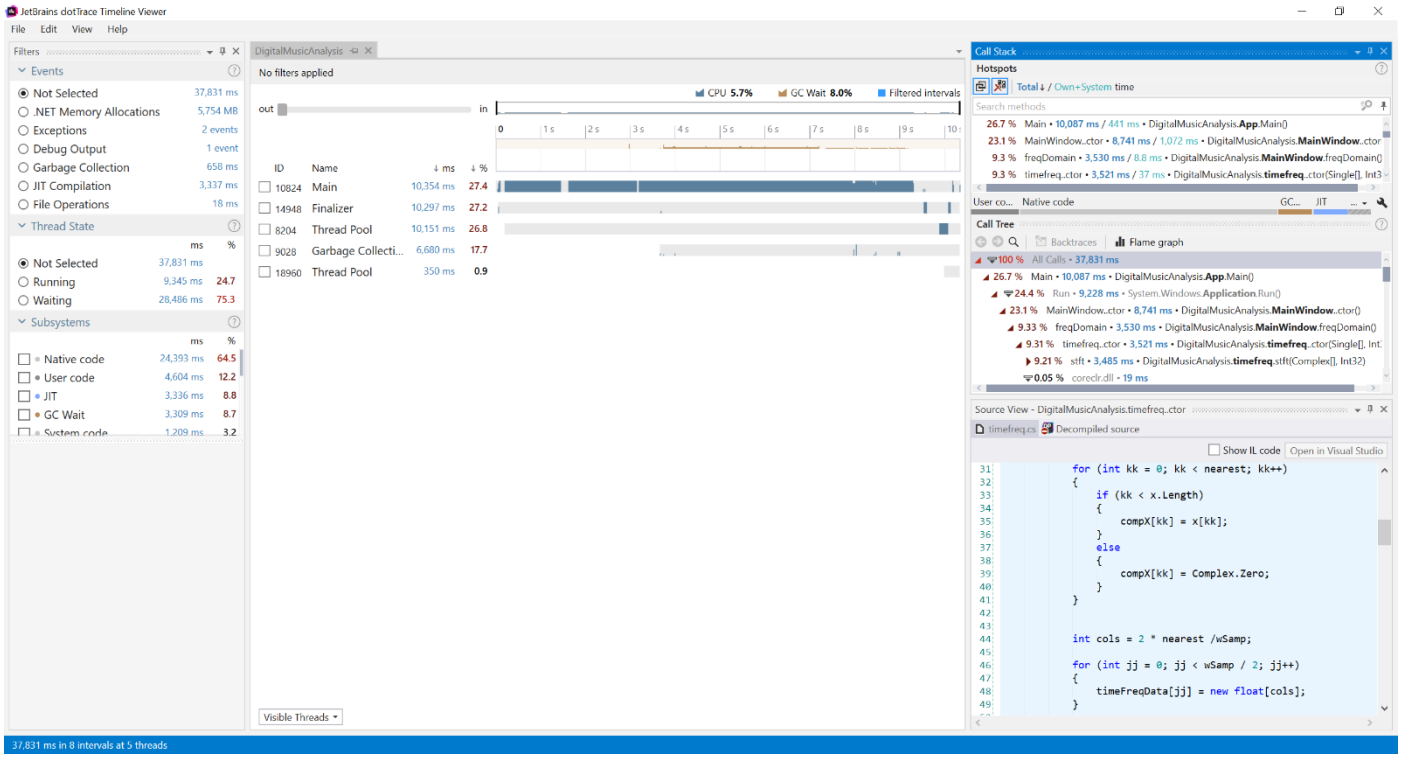


Figure 23 - original application's performance

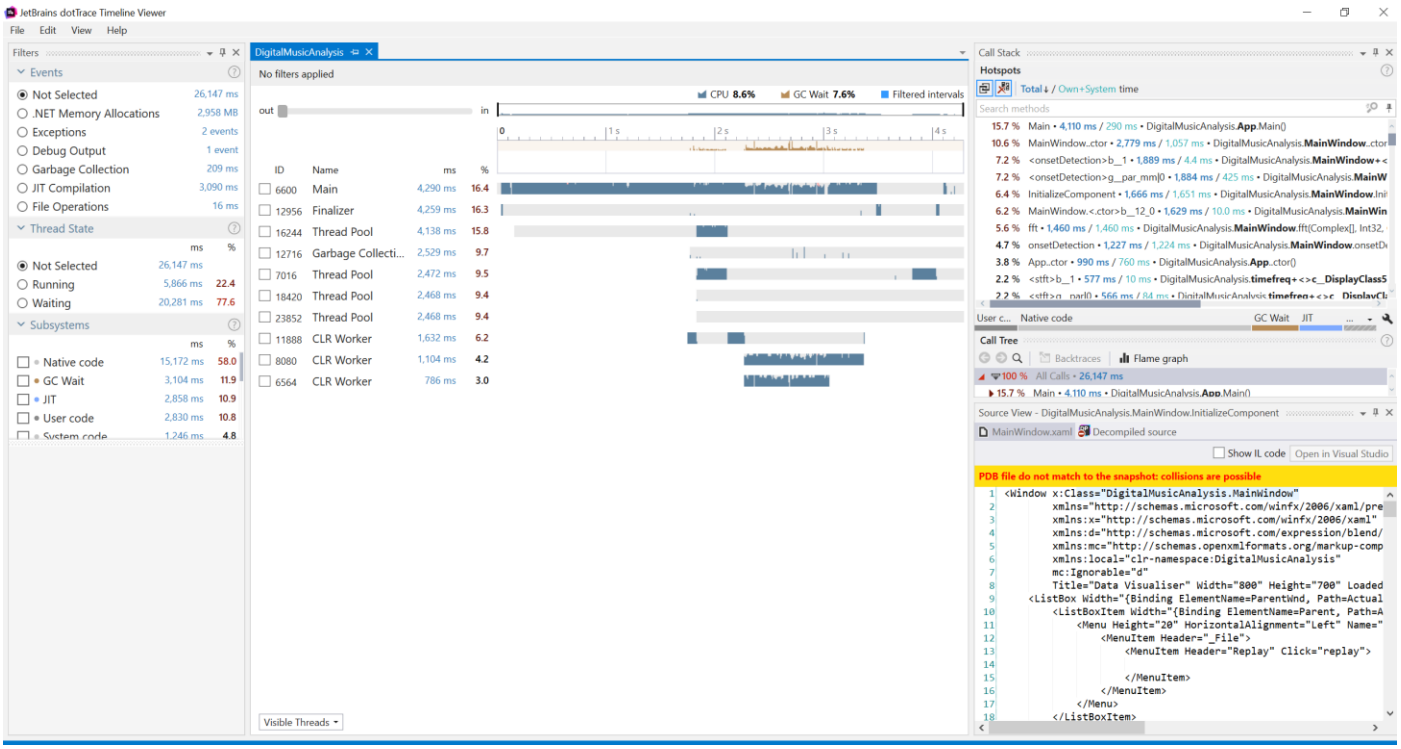


Figure 24 - parallel application running on 2 cores performance profiler

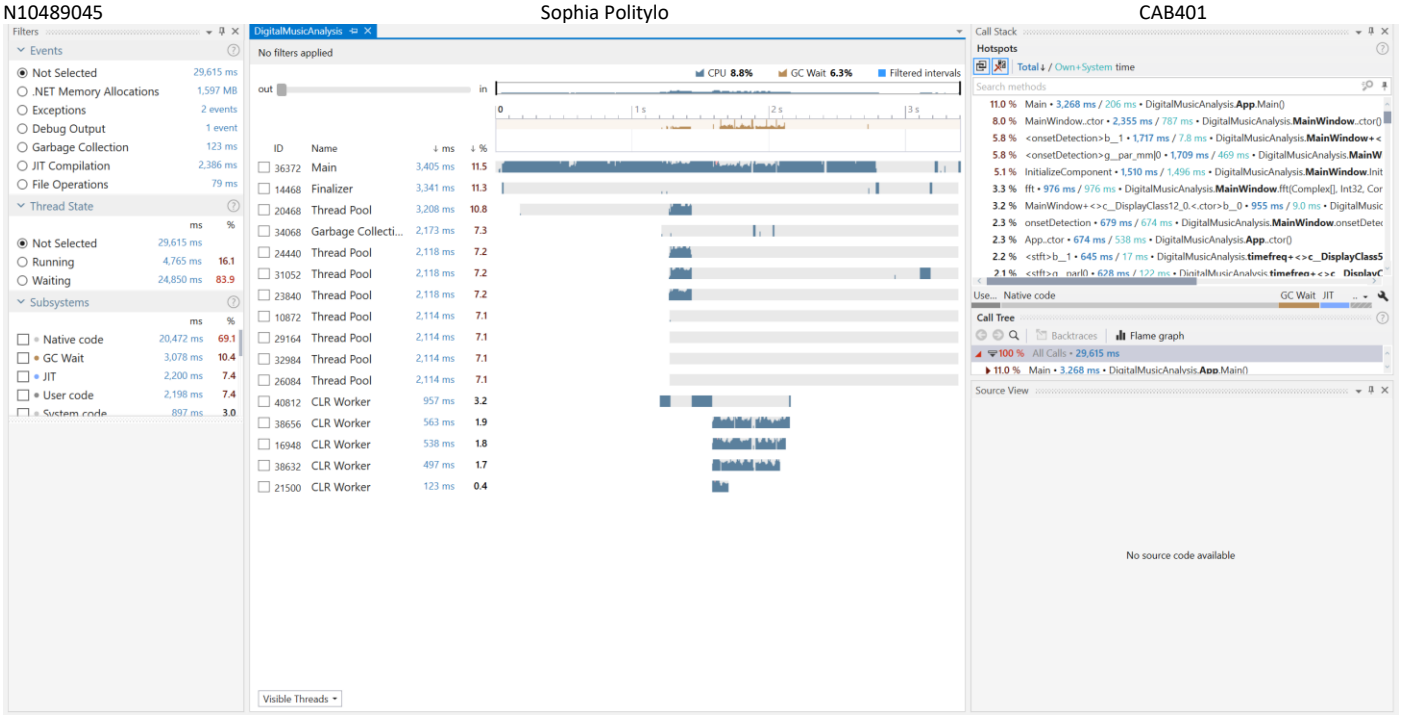


Figure 25 - parallel application running on 4 cores performance profiler

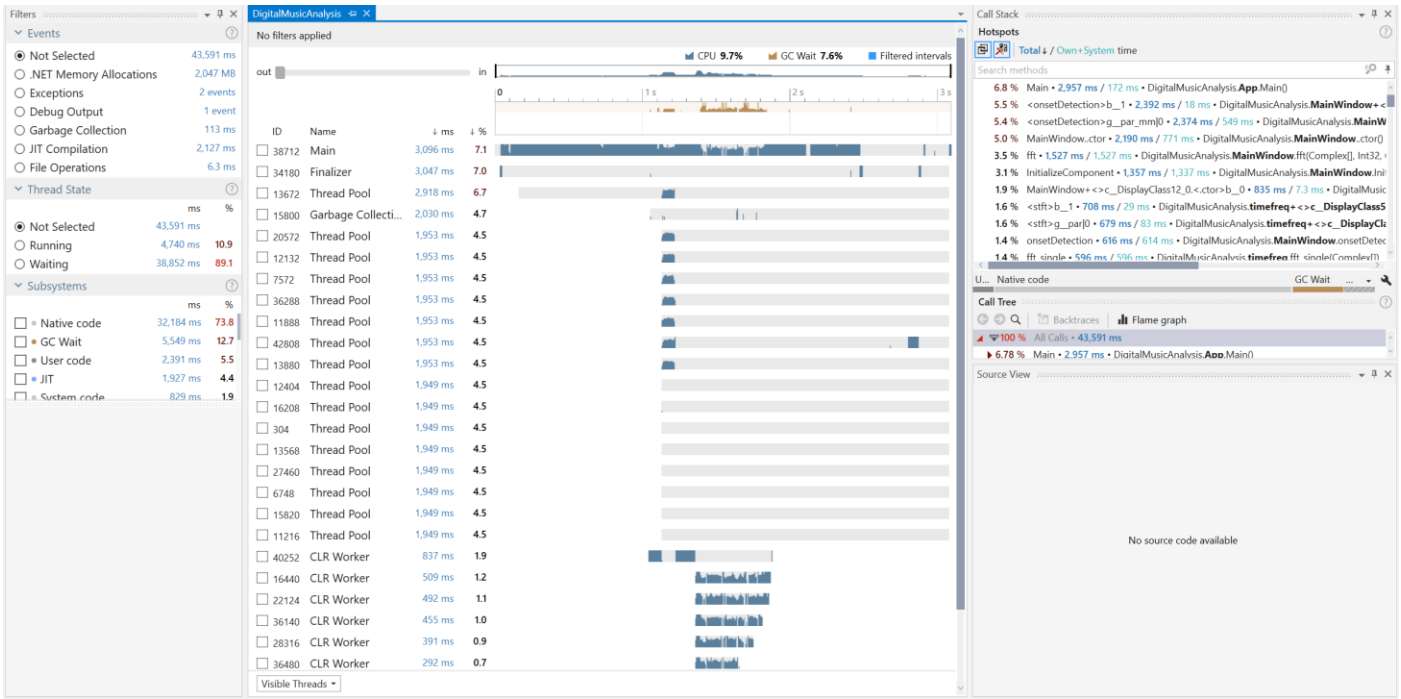


Figure 26 - parallel application running on 8 cores performance profiler

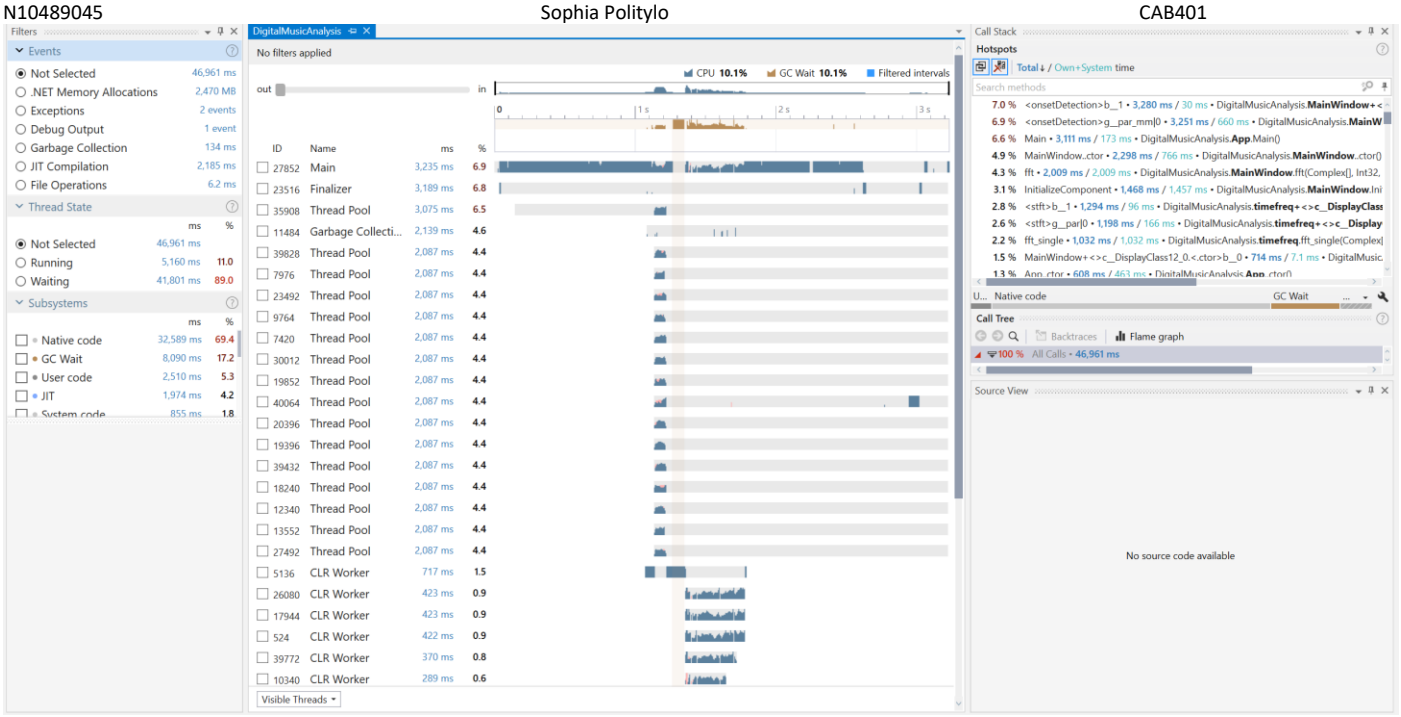


Figure 27 - parallel application running on 16 cores performance profiler

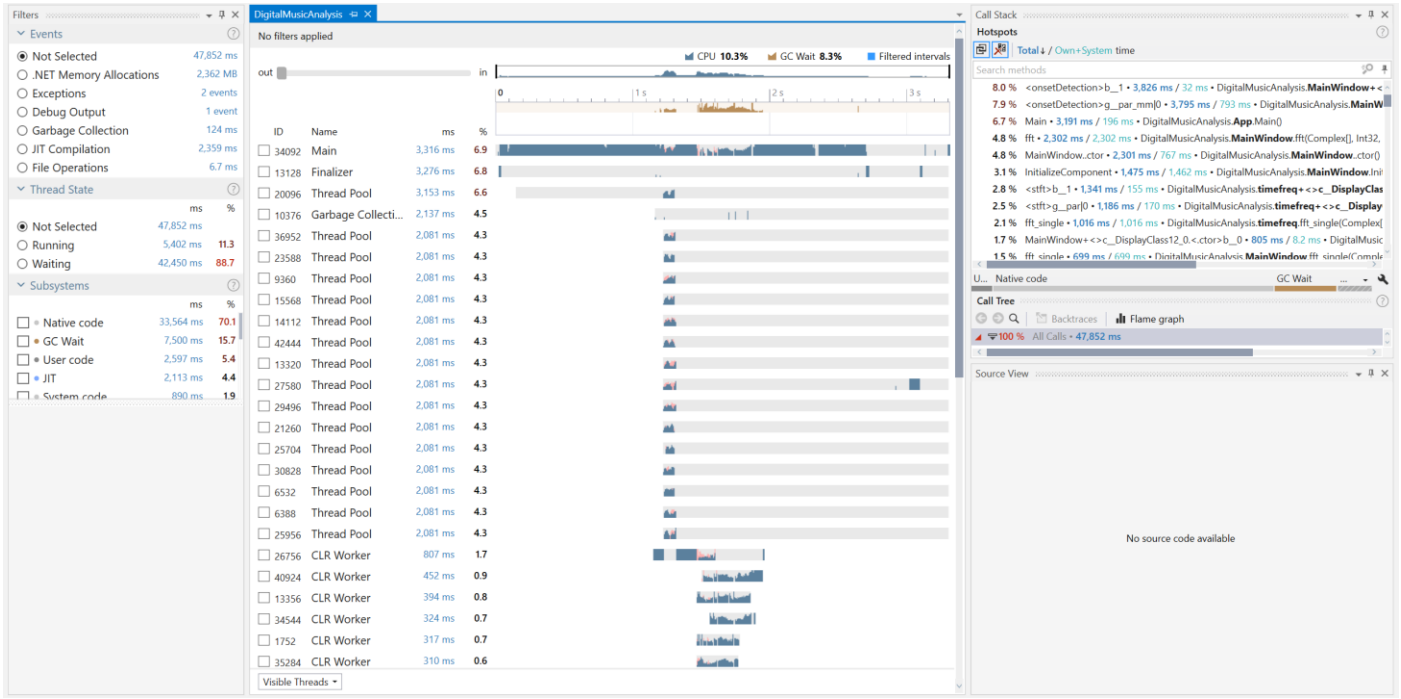


Figure 28 - parallel application running on 32 cores performance profiler

5.3.2 FFT performance

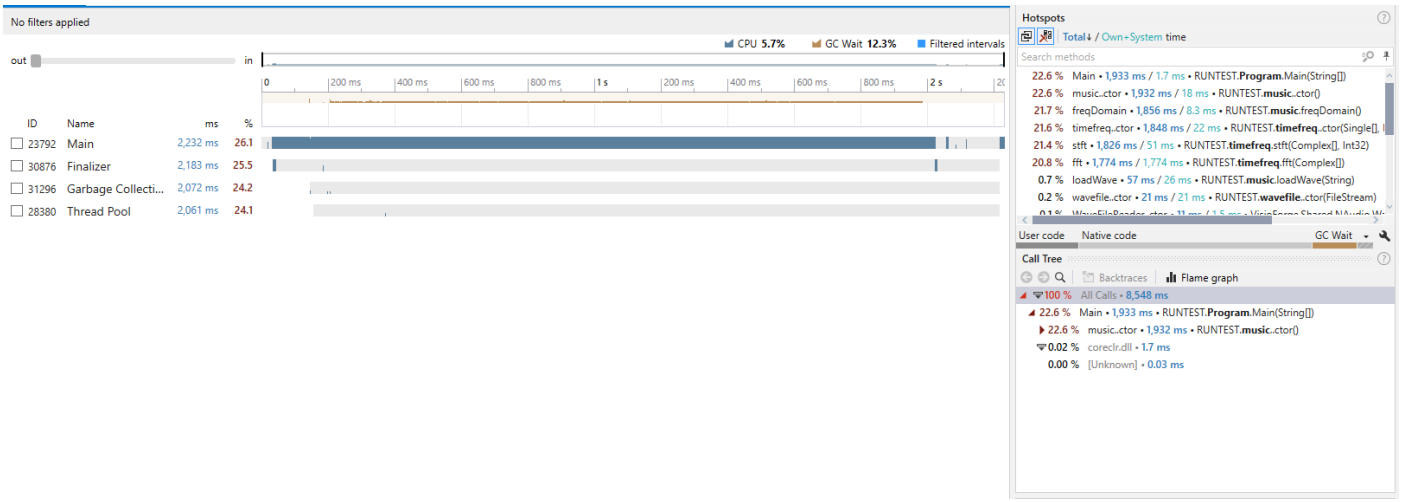


Figure 29 - original fft function performance profiler

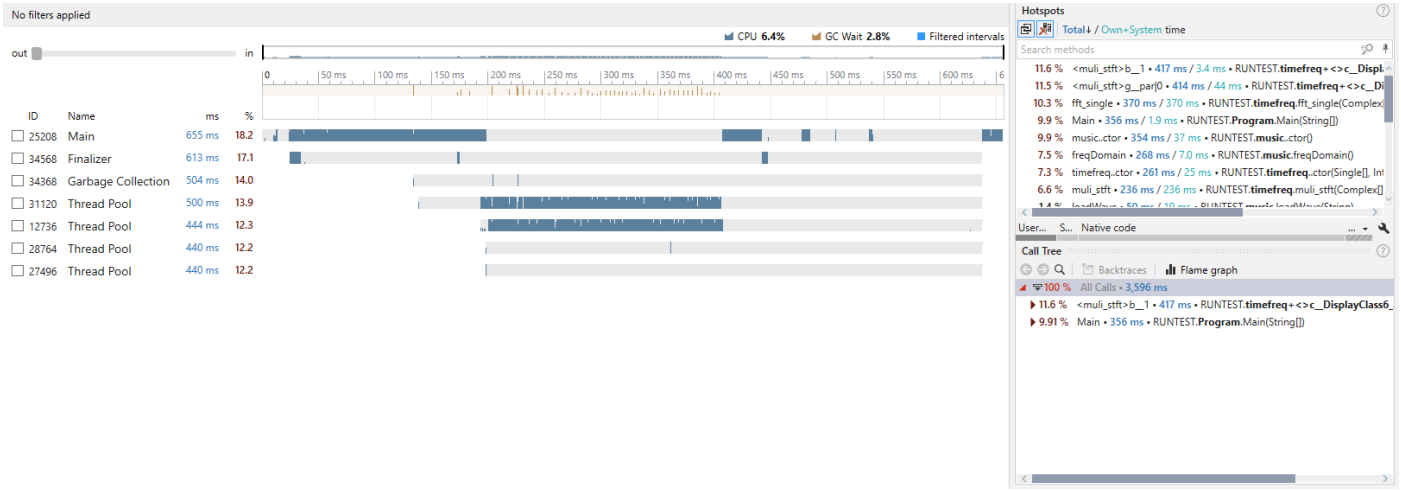
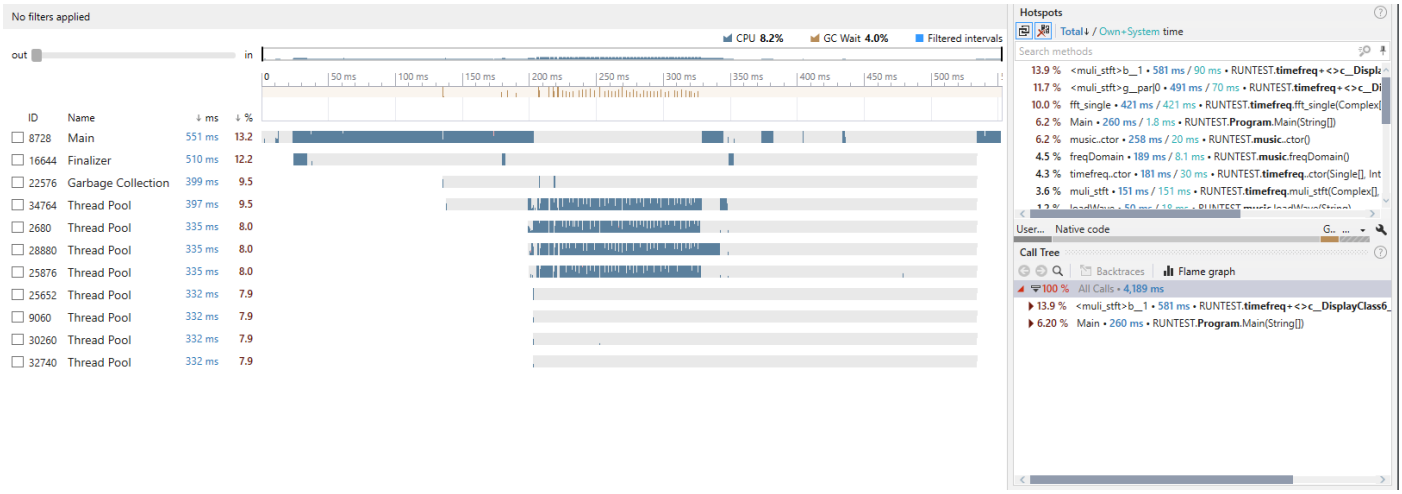


Figure 30 - new parallel fft function running on 2 cores performance profiler



new parallel fft function running on 4 cores performance profiler

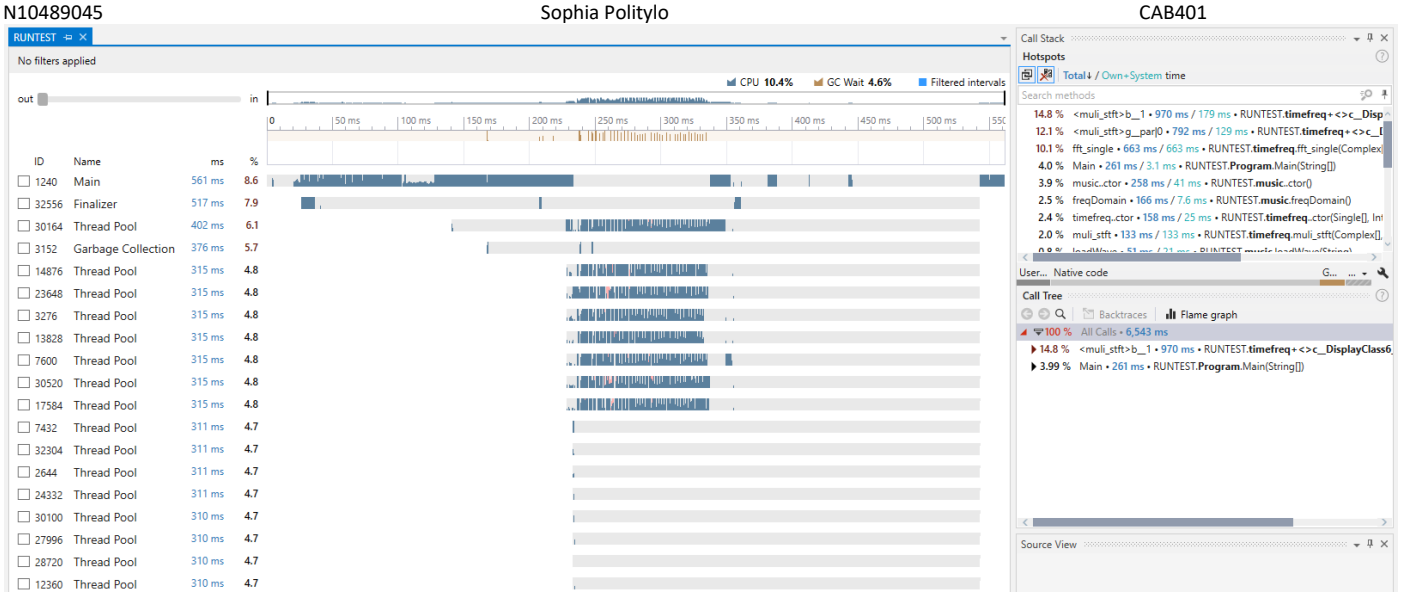


Figure 31 - new parallel fft function running on 8 cores performance profiler

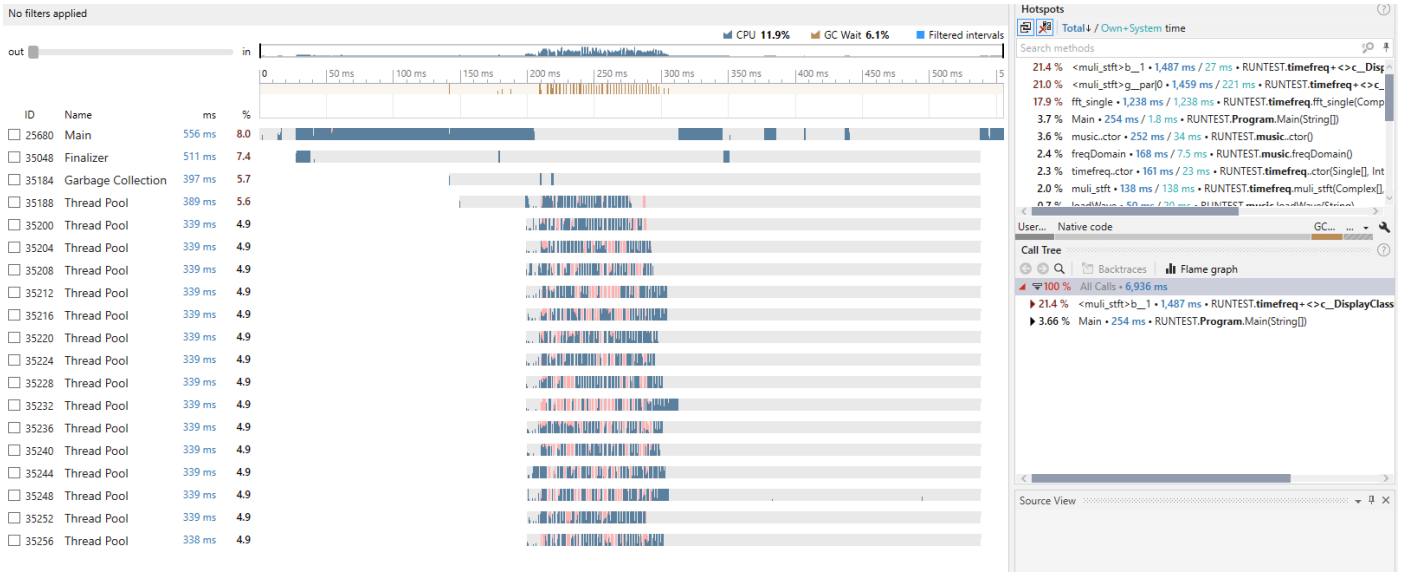


Figure 32 - new parallel fft function running on 16 cores performance profiler

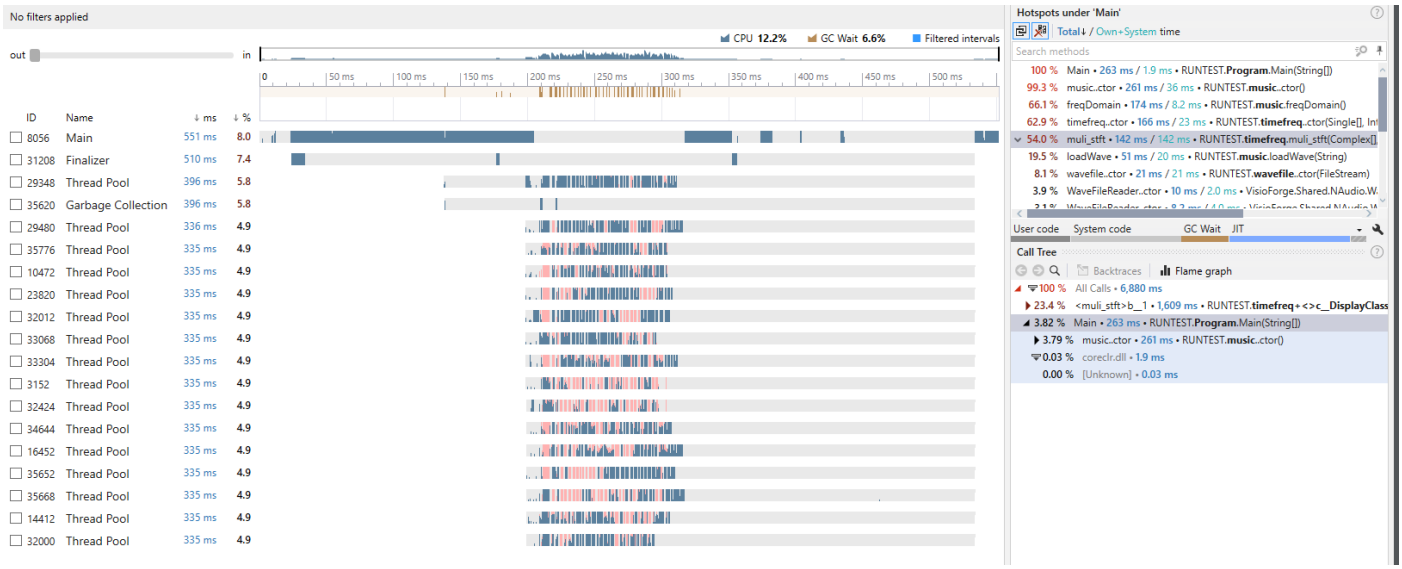


Figure 33 - new parallel fft function running on 32 cores performance profiler