Due:
28/10
2022

Digital Music Analysis

MULTITHREADING AND PERFORMANCE

THOMAS PETERS – N10810251

Hardware

CPU	
Intel Core i5-9600K (Coffee Lake-S) 6 Physical Cores, 6 Virtual Cores L1 Cache: 6x32 KB Instructional, 6x32 KB Data L2 Cache: 6x256 KB L3 Cache: 9MB	
TLB (Instruction)	TLB (Data)
2 MB/4 MB Pages Fully Associative 8 Entries	4 KB Pages 4-way set Associative 64 Entries

Memory
16GB Dual-Channel (2 x 8GB), 1333MHz

Sequential Analysis

Prior to analysis it is essential that the operation of the application from a black box standpoint is discussed. The chosen application is the Digital Music Analysis provided by QUT for the purpose of this assignment.

From an external viewpoint the application serves the primary function of analyzing an audio file, comparing it to an XML file that represents the relative sheet music, and providing that information back to the user in as a spectrogram and music notation. The music notation portion of the application displays both the expected and recorded notes and their relative information with comments.

A more in-depth analysis of the application reveals two primary components, onset detection and a large-scale time frequency converting class. Both the former and the latter implement a recursive fast Fourier transform (*RFFT*) for deconstructing the incoming wave into its base frequencies. Whilst reliable and fast, running in $N\ log_2 N$, the recursive nature mandates an excessive number of allocations with each recursion allocating four new complex arrays of $N/2$ to be processed. This means that from the highest level (the initial call) the allocated objects persist through recursion, resulting in bloating of the heap and increasing garbage collection events.

This is observable in *Figure A.3*. The yellow highlight is the Time Frequency class that handles the large-scale application of the RFFT, whilst the red represents the period in which on-set detection implores the

RFFT as well. As shown, the number of Generation 0 collection events that occur rises to well over 350 during its application whilst a new Generation 1 event occurs every couple of RFFT recursions.

A complete version can be found at with the submission.

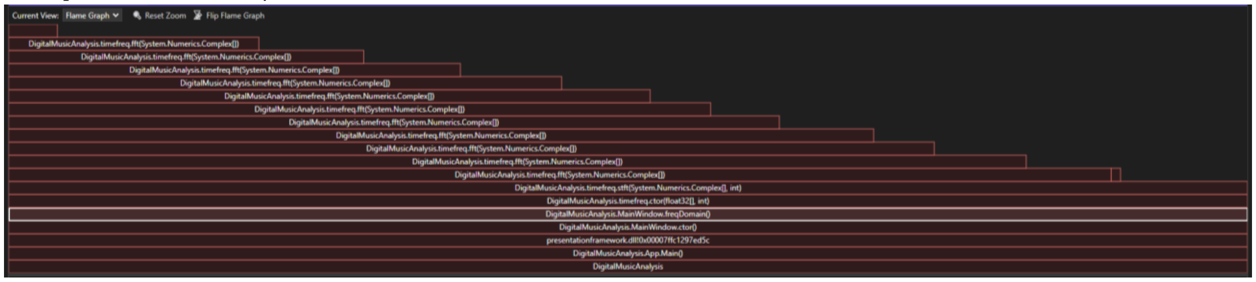

Current View: Flame Graph
Reset Zoom
Flip Flame Graph
DigitalMusicAnalysis.timefreq.fft(System.Numerics.Complex[])
DigitalMusicAnalysis.timefreq.stft(System.Numerics.Complex[], int)
DigitalMusicAnalysis.timefreq.ctor(float32[], int)
DigitalMusicAnalysis.MainWindow.freqDomain()
DigitalMusicAnalysis.MainWindow.ctor()
presentationframework.dll!0x00007ffc1297ed5c
DigitalMusicAnalysis.App.Main()
DigitalMusicAnalysis

Figure A.1: Timefreq Class FFT

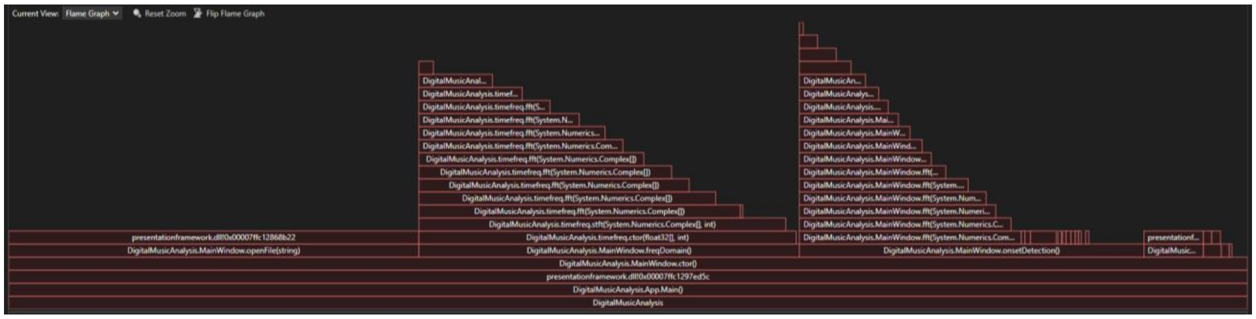

Current View: Flame Graph
Reset Zoom
Flip Flame Graph
presentationframework.dll!0x00007ffc12868b22
DigitalMusicAnalysis.MainWindow.openFile(string)
DigitalMusicAnalysis.timefreq.stft(System.Numerics.Complex[], int)
DigitalMusicAnalysis.timefreq.ctor(float32[], int)
DigitalMusicAnalysis.MainWindow.freqDomain()
DigitalMusicAnalysis.MainWindow.onsetDetection()
DigitalMusicAnalysis.MainWindow.ctor()
presentationframework.dll!0x00007ffc1297ed5c
DigitalMusicAnalysis.App.Main()
DigitalMusicAnalysis

Figure A.2: Full Application Fire graph

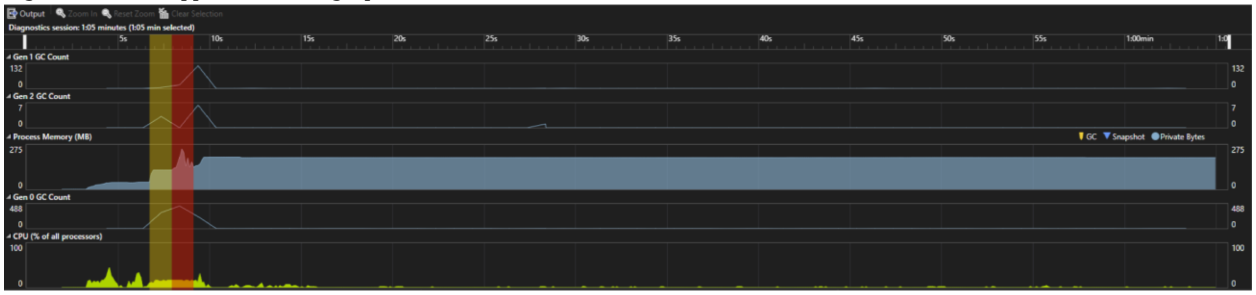


Figure A.3: Periods where RFFT is implemented

An additional point of worry also surrounds the allocation of data per need. For example, the allocation of *comp* for use with the FFT, following which, is collected by garbage due to the allocation of a new resulting array. This, whilst fine as a RAW dependency, introduces additional overhead as it performs the writing of data to a new local, prior to reading, adding to the memory usage.

Potential Parallelism & Efficiencies

Parallelism

The two mentioned points of performance loss extend primarily from the implementation of the RFFT. This is due to the sequential nature of the data as it mandates an in-order input and output that may not be

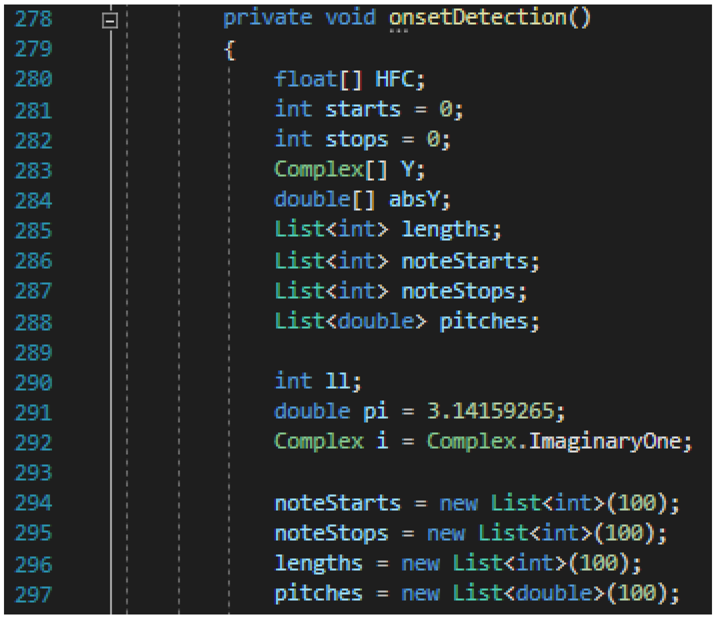
reliable in a multi-threaded design. However, the benefit of this is the granularity, as in essence the application employs nested looping to handle the mathematics. This means that the application could experience a notable performance benefit if redesigned to parallelize these loops, whilst also applying a more suitable FFT algorithm.
This would be a notable redesign however, as the FFT is RAW data dependent on pre-processed data that would need to be localized to the thread or would need to be thread-safe in usage. An example of this is "twiddles" found in *timefreq.cs* which is necessary to the functionality of the FFT algorithm and is compiled at line 18-23.

Fortunately, within *timefreq.cs* the twiddles are only read when applied to the algorithm, making it suitable for shared memory access between several threads. However, the implementation within *onsetdetection* found within the *MainWindow* class (line 278) parallels this, changing size relative to the current note and would require either a two-dimensional array to represent the notes index and twiddles if pre-compiled or would be handled locally within the thread.

Additionally, the style of multithreading will also have a significant impact on the speed of the application. For example, the overhead of creating $\lceil \frac{N}{w} \rceil * w$ threads where *w* represents sample size, may result in degraded performance in contrast to that of various continuous thread processing $\frac{\lceil \frac{N}{w} \rceil * w}{T}$ simultaneously.

Efficiencies

OnSetDetection employs numerous forms of data storage that would impact the use of spatial locality within the program. Specifically, lines 281-297 shown in *Figure B.1*, where the allocation of each portion of data representing a note may affect the ability to cache the relative notes additional data. This can be negated by first culling unnecessary values and implementing a value type variable capable of storing the relative data such as a Vector4 or custom struct. This would present improved spatial locality during further processing as each contained value would be caught within the structure itself.



```
private void onsetDetection()
{
    float[] HFC;
    int starts = 0;
    int stops = 0;
    Complex[] Y;
    double[] absY;
    List<int> lengths;
    List<int> noteStarts;
    List<int> noteStops;
    List<double> pitches;

    int ll;
    double pi = 3.14159265;
    Complex i = Complex.ImaginaryOne;

    noteStarts = new List<int>(100);
    noteStops = new List<int>(100);
    lengths = new List<int>(100);
    pitches = new List<double>(100);
```

Figure B.1

Abstractions & Implementations

For multi-threading the application the Task Parallel Library (TPL) was used. This is highly recommended in modern .Net applications and provides a quick method of making parallel programs and handles the bulk of provisioning previously done manually.
As the creation and destruction of threads is a time-consuming task, the TPL grants easy access to a premade pool of existing threads eliminating the additional overhead whilst granting a reliable method of multi-threading.
Additionally, beyond the use of explicit tasks, TPL provides access to parallel loops, which performs the separation and allocation of work to the tasks it creates.

Both forms of tasks (parallel loop and explicit) were used within the application at optimal times. For example, a parallel loop runs synchronously whilst providing parallel execution, whilst explicit tasks provide asynchronous parallel execution.

Because of this distinction, parallel loops were employed within *onsetdetection* as it presents a fixed quantity of work in the form of notes which allows TPL to allocate the maximum number of available tasks. *STFT* implements a fixed number of target tasks, each functioning continuously, removing the need for workload allocation or balancing, but introducing the need for synchronization when returning the results.
To ensure the resulting output maintains order, each workload includes its corresponding index, avoiding the possibility of overwriting; however, to ensure this never happens each thread performs a lock prior.
Lock was chosen as the overhead from system wide locking provided by mutex was substantial in comparison.

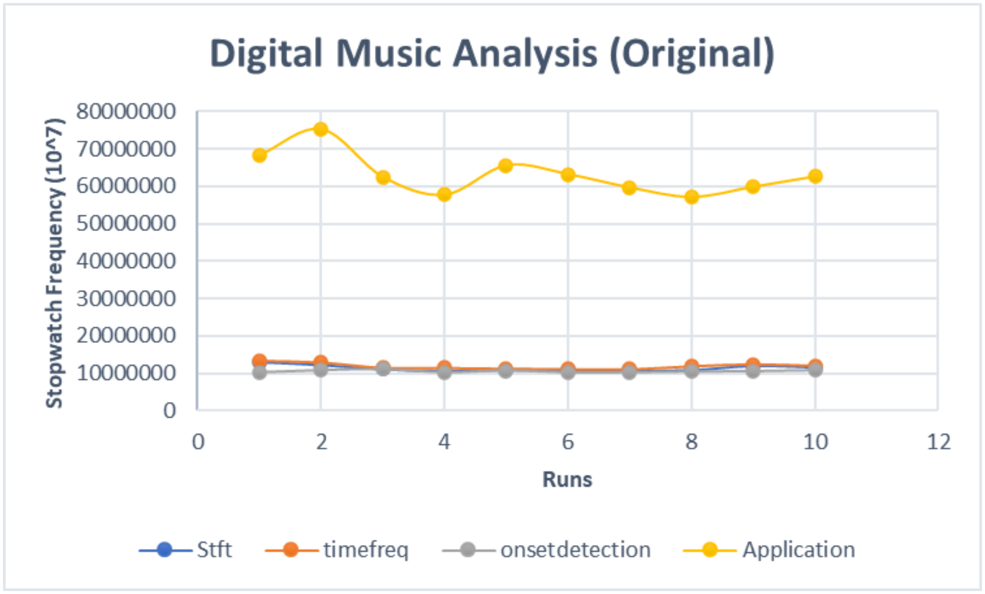
Results

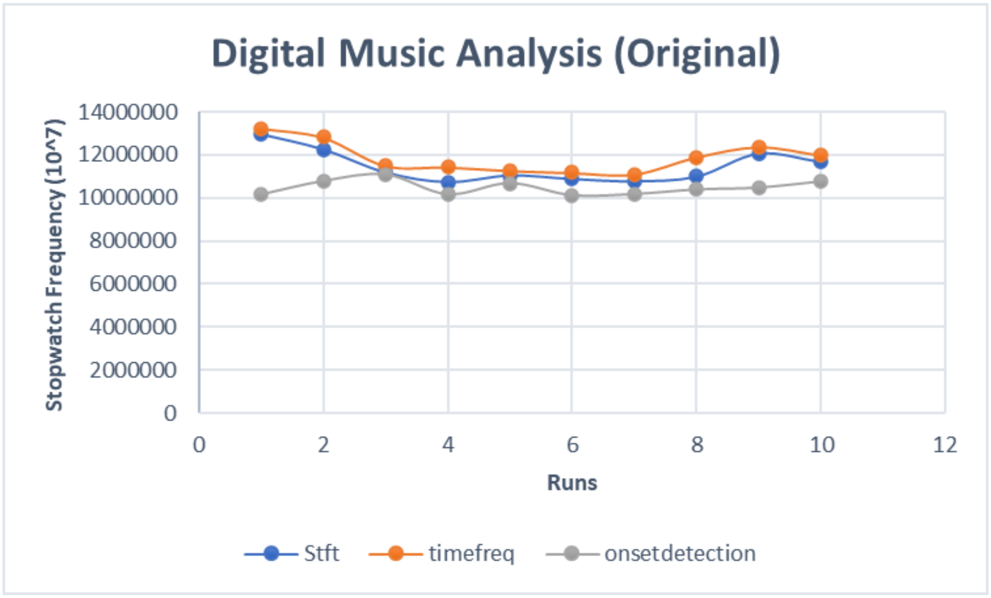
All diagnostic sessions are included in the submission

The results presented are taken from within the function they are present in and are measured using the diagnostic stopwatch in ticks per second. The portion measured in OnSetDetection was the beginning of the method, to the start of the staff tab display, and thus only reflects the impact of the note analysis portion of the application.

Sequential

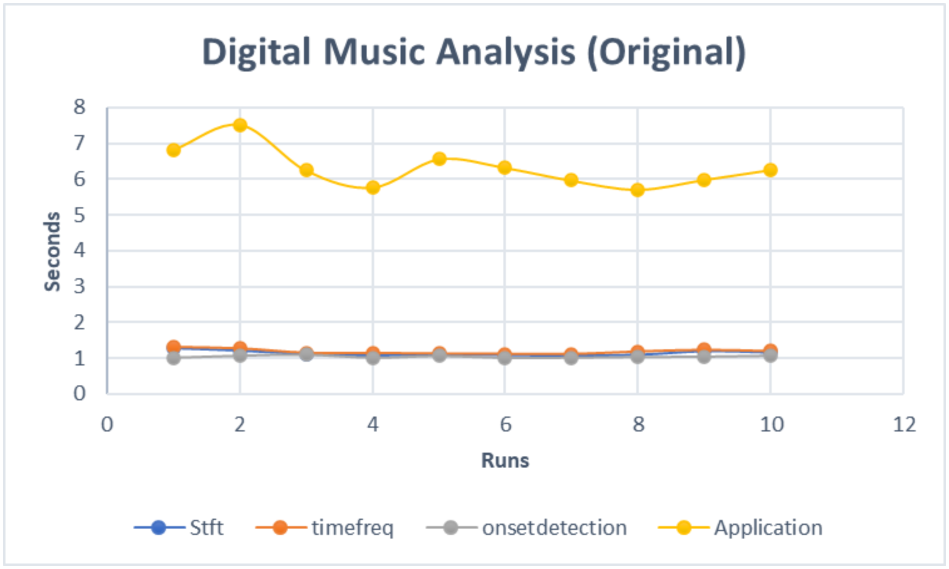
Ticks

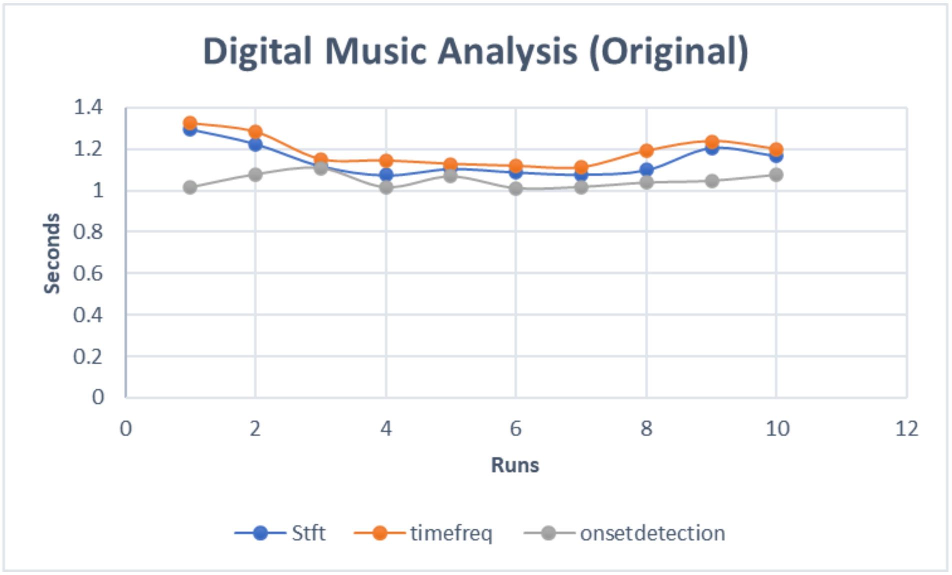




Raw Frequency										
	1	2	3	4	5	6	7	8	9	10
Stft	12942539	12242419	11188622	10744285	11045953	10885608	10778307	10989338	12043727	11665435
timefreq	13235683	12816172	11504403	11436214	11268257	11171479	11104780	11895901	12374088	11989791
onsetdetection	10158077	10778851	11084365	10169754	10680436	10121742	10180254	10388830	10475943	10761143
Application	68276796	75199409	62421350	57729386	65601924	63203609	59635143	57130682	59849890	62602261

Seconds



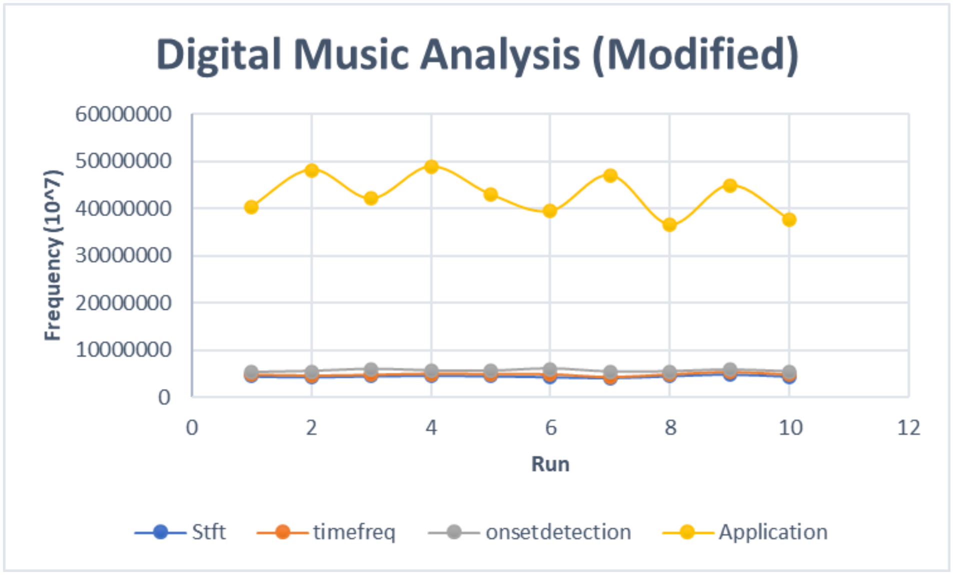


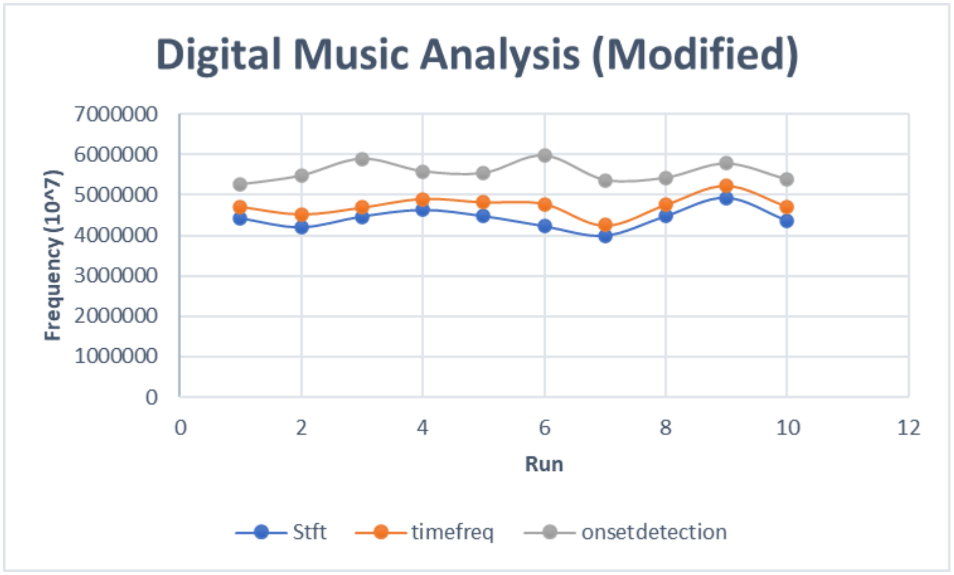
Seconds										
	1	2	3	4	5	6	7	8	9	10
Stft	1.2942539	1.2242419	1.1188622	1.0744285	1.1045953	1.0885608	1.0778307	1.0989338	1.2043727	1.1665435
timefreq	1.3235683	1.2816172	1.1504403	1.1436214	1.1268257	1.1171479	1.110478	1.1895901	1.2374088	1.1989791
onsetdetection	1.0158077	1.0778851	1.1084365	1.0169754	1.0680436	1.0121742	1.0180254	1.038883	1.0475943	1.0761143
Application	6.8276796	7.5199409	6.242135	5.7729386	6.5601924	6.3203609	5.9635143	5.7130682	5.984989	6.2602261

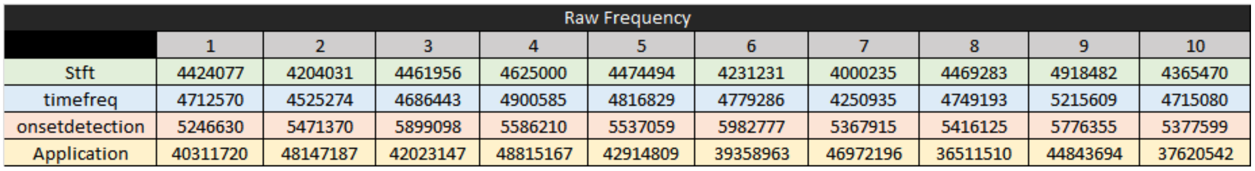
Parallel

With a target of 11, Radix-2 Consumers for Timefreq

Ticks

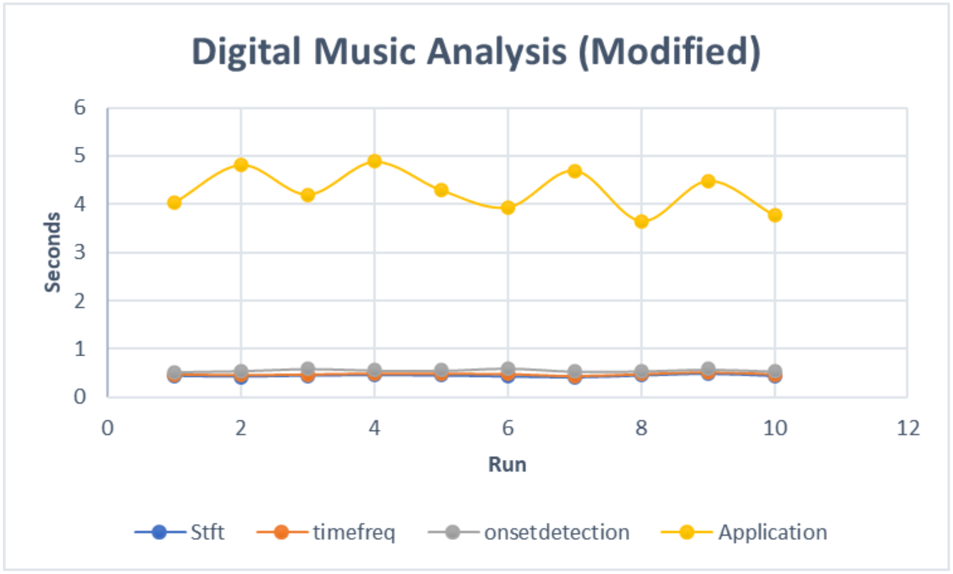


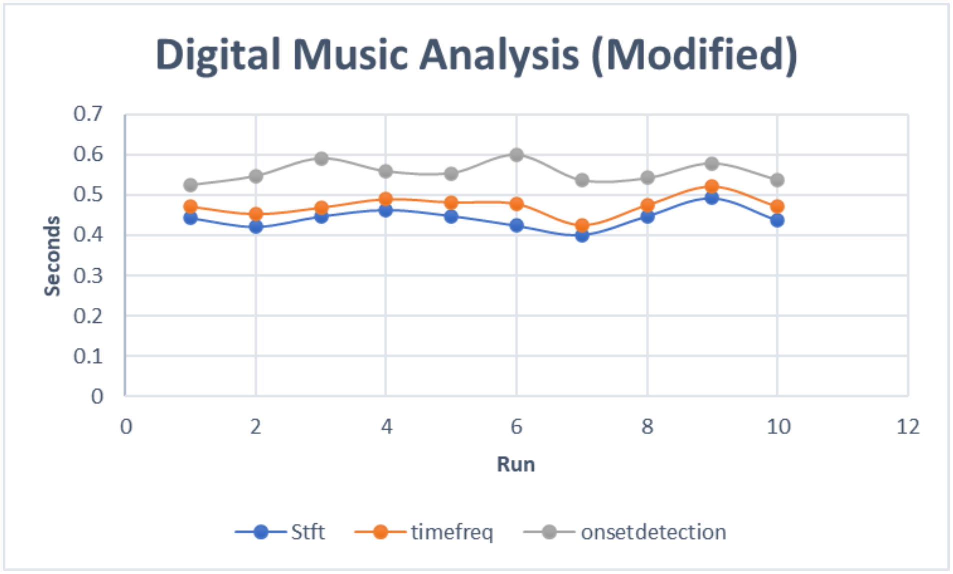




Raw Frequency										
	1	2	3	4	5	6	7	8	9	10
Stft	4424077	4204031	4461956	4625000	4474494	4231231	4000235	4469283	4918482	4365470
timefreq	4712570	4525274	4686443	4900585	4816829	4779286	4250935	4749193	5215609	4715080
onsetdetection	5246630	5471370	5899098	5586210	5537059	5982777	5367915	5416125	5776355	5377599
Application	40311720	48147187	42023147	48815167	42914809	39358963	46972196	36511510	44843694	37620542

Seconds





Seconds										
	1	2	3	4	5	6	7	8	9	10
Stft	0.4424077	0.4204031	0.4461956	0.4625	0.4474494	0.4231231	0.4000235	0.4469283	0.4918482	0.436547
timefreq	0.471257	0.4525274	0.4686443	0.4900585	0.4816829	0.4779286	0.4250935	0.4749193	0.5215609	0.471508
onsetdetection	0.524663	0.547137	0.5899098	0.558621	0.5537059	0.5982777	0.5367915	0.5416125	0.5776355	0.5377599
Application	4.031172	4.8147187	4.2023147	4.8815167	4.2914809	3.9358963	4.6972196	3.651151	4.4843694	3.7620542

Based on the averages of each Frequency test

Performance %			
Stft	timefreq	onsetdetection	Application
61.42869817	60.14049473	46.88792049	32.3171645

Result tests

For result testing I originally performed content comparison on the sequential values and parallel values. However, during this testing phase the bulk of the Radix-2 FFT was developed in an individual project, which received the inputs given to the Recursive FFT through a text file. Unfortunately, during this I discovered the twiddles generated did not match despite the equation matching. After further research it became apparent that there is an acceptable degree in variation in results when applying an FFT to any input. Fortunately, the largest deviation I could find during the process began at $1 * 10^{-6}$ whilst most values experienced a variation at $1 * 10^{-8}$. After hours of diagnosing, I was unable to find the source of the problem, however, at such low deviation, I found it caused no notable errors when applied. Following this I resorted to evaluating the saved values against the generated ones through the use of a threshold a percentage.

After confirming the functionality was acceptable, the Radix-2 FFT was included into the application for a functional evaluation against the visuals presented in the original, and no observable variation was found.

Any further testing and analysis were completed through the use of the visual studio performance profiler.

Performance Barriers

When analyzing the application initially, I intended to produce several threads that would each handle a portion of data. However, it quickly became obvious that the overhead produced was detrimental to the goal of performance. Instead, the implementation with timefreq shifted towards an indexed Producer/Consumer pattern.
Unfortunately, however, the full performance of this layout isn’t utilized in the modified version as the production of workloads has remained sequential on the main thread and is performed prior to the creation of the consumer tasks, which are also sequentially created, making this a multi-threaded consumer design.
Furthermore, upon researching I determined that this format would be optimal for time frequency analysis, as it provides the ability to have a static set of tasks, without the need for segment creation. As the output result is required in-order, the chosen storage was a jagged array. This was done as jagged arrays at scale are faster than a multi-dimensional array in exchange for memory consumption, a downside generated by the .Net Runtime that is being improved in .Net 7.

Code Breakdown

This will exclude general edits such as stopwatches for results, and general clean up.
Additionally, these line references refer to the modified application and the purpose within the modified enviroment.

Line	Source	Purpose
31	timefreq.cs	Calls GenerateLookup
22		Changed constructor to Task to allow for async
51		Changed stft to await
73		Creation of ConcurrentQueue for workloads
73-84		Creation of workloads
86-100		Creation of Consumer tasks
133-195		Recursive FFT (edited) Consumer task and method
199-256		Radix-2 DIF Iterative FFT Consumer task and method
260-285		generateLookup method
287-317		In-place Bit reversal
131-132	musicNote	New constructors for Vector4 input
40-45	MainWindow	Moved all functions originally in the window constructor to an asynchronous method at lines 49-81
47		Task variable to artificially await
59		"await freqDomain()"
295		Removed constructor arguments from timefreq creation
296		"await stftRep.timefreq_init(waveIn.wave, 2048);"
312-337	onsetDetection	Removal of individual lists
421-469		Code was changed to use Vector4
473-543		Parallel for introduced to replace the original conversion of Lists to musicNotes
876-907	MainWindow	Radix-2 DIF Iterative FFT for use by onsetDetection
909-944		In p-place Bit reversal

Reflection

In hindsight there is a significant number of alterations I would like to make surrounding object creation. My initial plan was to reduce the number of Garbage collection events occurring and I find I have successfully met that task by replacing both instances of recursive fast Fourier transforms with an In-place Iterative Radix-2 algorithm that effectively reduced the excessive number of collection events occurring. During my initial testing, I found that the RFFT was consistently taking around 900ms in release after several runs whilst my Radix-2 implementation was reaching 500ms, and sometimes 450ms under the same conditions. However, using it inside of onsetdetection comes with a major downside courtesy of the required bit reversal. Due to the varying duration of each note, the option to generate a lookup table is pointless as the period required changes, the number of bits required within the reversal also fluctuates. Unfortunately, because of this the largest portion of onsetdetection's runtime is made up of performing bit reversals despite still functioning faster than its recursive counterpart. This is mainly due to the way I have performed the reversal. Currently, I convert the integer to a string, pad with the required zeros and loop through each character before converting back into an integer. While fast enough to generate a static lookup table once, this results in the process taking additional time due to nesting. This could be mitigated using Bitwise operations; however, I could not come up with a process of reversal within the designated number of bits without using a loop.

Furthermore, I regret not dedicating more time to other components of the application. Whilst Winforms was a staple of simple and quick UI creation, its runtime being handled within the main thread can easily be a drawback when working within the environment. With this knowledge I would've additionally liked to investigate the possibility of wrapping both I/O related functions for reading notation and audio, in their own tasks/threads to perform simultaneously as this could have resulted in a minor improvement in the applications total time, but overall would not have been a significant impact.
Another possible point of improvement is with the drawing of notation. Whilst the existing design handles it using various loops, I believe that introducing the UI portion of the notation into the MusicNote class to be stored with related data would perform better as the flow of the application would shift from sequential loops to generate the components, towards applying pre-existing data written during the initial creation of each note.

In various cases I replaced variables used for outputs with re-assignment of the input values. I am unsure if this in any way would impact performance as the memory has already been allocated for the initial operation, or if this would classify as a WAW or RAW data dependency. Furthermore, I find that whilst inline usage of stopwatches functioned for acquiring the results presented, I would much rather have performed the task of benchmarking through the use of Benchmarkdotnet, however due to my unfamiliarity with Winforms, I failed to make Benchmarkdotnet successfully perform any benchmarking.

Ultimately, I believe I have successfully completed my goal; however, I acknowledge that my implementation, whilst functional and reliable, is far from perfect and would require more refinement. This solely extends from my lack of familiarity with the application on a fundamental