David Mai

N11252502

Parallel Computing

CAB401 A2

Contents

[1.0 Introduction: 2](#_Toc180844990)

[1.1: Digital Music Analysis Application 2](#_Toc180844991)

[1.2 Program implementation 2](#_Toc180844992)

[1.3 Key Components: 2](#_Toc180844993)

[2.0 Analysis of Potential Parallelism 2](#_Toc180844994)

[2.1 Potential Parallelism Opportunities 2](#_Toc180844995)

[2.2 Data Dependencies 3](#_Toc180844996)

[2.3 Scalability and Granularity 3](#_Toc180844997)

[2.3 Changes Required for Parallelism 3](#_Toc180844998)

[3.0 Mapping Data to Processors 4](#_Toc180844999)

[3.1 How computation data were mapped to processors 4](#_Toc180845000)

[3.2 Parallelism Abstractions 4](#_Toc180845001)

[4.0 Results 4](#_Toc180845002)

[4.1 Result Explanation 6](#_Toc180845003)

[5.0 Sequential vs Parallelised Accuracy 7](#_Toc180845004)

[6.0 Compilers and tools used 7](#_Toc180845005)

[7.0 Overcoming barriers 7](#_Toc180845006)

[7.1 Issues 7](#_Toc180845007)

[8.0 Parallelisation Code Explanation 8](#_Toc180845008)

[9.0 Reflection and Conclusion 8](#_Toc180845009)

[Appendix 9](#_Toc180845010)

# 1.0 Introduction:

## 1.1: Digital Music Analysis Application

The Digital Music Analysis application was initially developed to assist beginner violin players by providing real-time feedback when practicing without a teacher. The application analyses audio input compares the player's performance with predefined sheet music and identifies any pitch or timing errors. By utilising audio signal processing techniques, such as Fourier Transform, the application can extract and analyse musical features to help users improve their playing skills. This report discusses the parallelisation of the application to enhance its performance, focusing on identifying areas for optimisation, implementing parallel programming techniques, and assessing the outcomes of the parallelisation effort. The introduction should clarify that this application was originally developed to assist beginner violin players by providing real-time feedback when no teacher is present.

## 1.2 Program implementation

The original application was developed to analyse digital music files by detecting musical notes, determining their frequencies, and displaying the information visually. The application takes a .wav and .xml file as input, processes it to detect musical events, and then visualizes both time-frequency information and the musical score. The analysis relies on several key computational steps, including Short-Time Fourier Transform (STFT) to generate a time-frequency representation of the input waveform, onset detection to identify the beginning and end of musical notes, and pitch analysis to determine the frequency of each detected note.

## 1.3 Key Components:

The architecture of the original application can be divided into several key components and workflows. These include MainWindow, which serves as the primary user interface for the application. It ties together all functionality which includes handling user inputs such as loading audio files, processing analysis functions, and displaying the processed results. This component is critical as it orchestrates the interaction between the user and computational logic of the application. MainWindow also manages controls and visual elements that allow the user to see the analysis results in real time. Timefreq class computers the Short-time Fourier Transform (STFT) of the audio signal and stores the results in a matrix representation.

# 2.0 Analysis of Potential Parallelism

## 2.1 Potential Parallelism Opportunities

The application contains several areas where parallelism can be introduced to improve performance, especially considering its computational nature involving Fast Fourier Transforms (FFT), time-frequency analysis, and music note extraction. The following analysis identifies existing opportunities for parallelism and evaluates their granularity, scalability, and safety. The application already implements parallelism in some areas, such as FFT computation, bit-reversal reordering, and Short-Time Fourier Transform (STFT) calculations. However, additional opportunities for parallelism can be found throughout the code, particularly in loops and repetitive tasks involving signal processing. Below are the key areas where parallelism is or can be introduced. These include bit-reversal reordering (in FFT.IterativeCTFFT) which involves rearranging elements in a bit-reversed order. This involves using the Cooley-Tukey FFT algorithm which involves multiple layers of computations where each element in the array is processed independently to which Parallel.For() then divides each layer across multiple threads thus efficiently utilising the available cores to perform calculations. The timefreq.stft method was also parallelised as it involves processing overlapping segments of the audio signal making it parallelisable. Finally the onsetDetection method which involves using a loop that calculates High Frequency Content (HFC) for each segment making the task parallelisable.

## 2.2 Data Dependencies

To determine which sections of the code were safe to parallelise, an analysis of data and control were considered:

* Bit Reversal reordering involves reordering elements in an array with each element’s new position being independent. Resulting in non-data dependencies that prevent prarllelism.
* Cooley-Tukey FFT algorithm requires results from the previous stage, introducing a control dependency. However within each stage, computations for different pairs of elements are independent.
* STFT involves overlapping windows of the input signal, but each window is processed independently, making it suitable for parallel execution.

## 2.3 Scalability and Granularity

The parallelism in STFT and onset detection scales well with them number of available threads, with both operations being able to effectively split across multiple processors, with minimal need for synchronisation. However, scalability is ultimately limited by factors such as memory bandwidth and overhead of creating and managing threads.

Both the OnSetDetection and FFT calculations also have computational cost per iteration, which makes them suitable for parallel computing. This level of granularity minimises the overhead associated with thread management and synchronisation issues.

## 2.3 Changes Required for Parallelism

To fully exploit the parallelism available in the application, the following changes were made:

1. Existing loops that operate on independent elements were modified to use Parallel.For() to exploit data parallelism in bit-reversal reordering, FFT computation and onset detection.
2. Thread pool tuning was also utilised usi9ng the ThreadPool.SetMinTHreads() method in MainWindow to ensure that a sufficient number of threads are available for the task. This not only helps maintaining balance between the overhead thread management, but also is crucial for collecting data on the task.
3. Algorithms were restructured such as the iterative Cooley-Tukey FFT algorithm which was adapted to allow each layer of computation to be performed in parallel, thus reducing overall computation time.

The parallelism introduced in this application targets the most computationally intensive tasks, such as FFT, STFT, and onset detection. These tasks are parallelized at a fine granularity, making them efficient for multi-core processors. The scalability of these tasks ensures that as the workload increases (e.g., longer audio files or higher-resolution analyses), the performance benefits of parallelism become even more significant. By analyzing data and control dependencies, the code was safely parallelized to maintain correctness while improving performance.

# 3.0 Mapping Data to Processors

## 3.1 How computation data were mapped to processors

Data and computations were mapped to processors using the Parallel.For() construct from the .NET Task Parallel Library (TPL). This library does all the heavy lifting of complex thread management, which allows the developer to focus on specifying parallel tasks while TPL dynamically handles the allocation of work across available processing cores. Key aspects of mapping computation and data to processors include data decomposition which data is divided into independent chunks that could be processed simultaneously. This can be seen in the FFT computation where each stage of the operation was divided into parts that were computed concurrently. The TPL scheduler also dynamically assigns iterations of Parallel.for() to available threads thus balancing the workload to avoid any single core being overburdened while others are idle which ensures efficient use of all available threads.

## 3.2 Parallelism Abstractions

To achieve efficient parallel execution, the following parallelism abstractions and programming language constructs were used:

* Parallel.For() was used extensively to parallelise loops that involve independent iterations
* ThreadPool.SetMinThreads() was used to ensure that enough threads were available for handling parallel tasks. This was also used to pull data from the application to see how using less or more threads can impact the performance of the application.
* Implicit synchronisation at the end of each parallel loop ensures that no thread proceeds to the next stage until all iterations are complete which guarantees data consistency when moving from one phase to another.

# 4.0 Results

To evaluate the effectiveness of parallelisation, the use of stopwatch was used for both STFT and OnsetDetection. Here are the recorded times:

#### Figure 1: Sequential Data best times taken

|  |  |
| --- | --- |
| Tasks | Time Taken (ms) |
| STFT | 1733 |
| OnSetDetection | 1681 |

#### Figure 2: Parallelised Data

A table with numbers and text

Description automatically generated

#### Figure 3: Speed up Graph for STFT

#### Figure 4: Data for OnSetDetection

A screenshot of a data

Description automatically generated

#### Figure 5:Speed up Graph for OnSetDetection

## 4.1 Result Explanation

The sequential implementation for the STFT took 1733ms with no modifications to the algorithm. OnsetDetection took 1681ms. After parallelisation both STFT and OnSetDetection shows significant improvement between 1-6 threads being utilised. The speed up graph shown above illustrates the improvements of parallelisation with it plateauing from 6 threads to 16 threads which indicates possible overhead or conflict between the threads.

# 5.0 Sequential vs Parallelised Accuracy

To test that the parallel version of the application produced the same results as the original, there were two simple approaches that were employed to ensure correctness. The first being a visual comparison of the output. As shown in appendix 7-11 were the spectrograms and musical notes which were visually compared between both sequential and parallel versions. The screenshots provided show identical visualisations, indicating that bother versions produced consistent outputs. The parallelised version produced matched those from the original sequential version. Appendix 11 also shows the second verification method to ensure accuracy of results. These statements print the alignment between deted notes and the expected notes from the musical score. By manually examining the output, this can be used to measure the accuracy and consistency of both version’s outputs.

# 6.0 Compilers and tools used

To parallelise the application effective, the following compilers, software, tools and techniques were used. The compiler was compiled using the C# compiler that comes with the .NET SDK. The compiler supports all the necessary features to handle parallel constructs and optimisations for multi-threading an application. The .NET Framework and Task parallel Library (TPL) was used as the foundation for the application which provided access to parallel programming using the Parallel.For() which managed threads efficiently. The IDE used was JetBrains Rider. It provided features such as code analysis, debugging, and built-in profiling tools such as dotTrace to help identify performance bottle necks and validate correctness of the parallel implementation. DotTrace also helped analyse execution times, identify hotspots, and evaluate the effectiveness of parallelisation by providing a detailed insight into thread behaviour. Thread pool configuration was used to configure the number of threads the application used which helped when generating data such as the speedup graph. The combination of these compilers, software, tools and techniques enabled the successful parallelisation of the application, resulting in substantial performance improvements whilst maintaining accuracy of the results.

# 7.0 Overcoming barriers

## 7.1 Issues

During the process of parallelising the application, several performance problems and barriers were encountered. These include load imbalance, granularity issues and the change of algorithm. In the initial stages of parallelisation, there was a noticeable load imbalance where some threads finished their work significantly earlier than others, leading to idle cores. To somewhat mitigate the issue, the workload was balanced using the TPL schedular, which was assigned iterations of Parallel.For() more evenly. Granuarility was also another issue as overly fine-grained parallelism led to excessive overhead from thread management, which negated the benefits of parallel execution. This was somewhat solved by adjusting the tasks so that each thread handled a sufficiently large chunk of data, thus balancing the CPU’s workload.

# 8.0 Parallelisation Code Explanation

The modifications and additions made to parallelise the application were made within 3 files. MainWindow.xaml.cs, timefreq.cs, and the addition of ftt.cs. Within MainWindow, line 48 and 51 the useParallel Boolean flag and an integer for specifiedThreadCount to specify the number of threads was used. Line 59 shows an example of ensuring that the thread pool has a minimum number of threads available to match the specified count, making sure that enough threads are available. FreqDomain method also includes this as shown in line 296 to ensure that the STFT calculations in timefeq.cs can take advantage of multiple threads during execution. From line 348 to 433 contains parallelisation for HFC calculations, twiddle factor calculations and FFT for OnsetDetection. The parallel version distributes the workload across multiple threads for efficiency. This parallelisation method significantly improved the speeds. The new fft.cs file also contains parallelisation modifications on both line 31 and 44. Line 31 allowed each index to be computed independently, which improved the reordering step for larger inputs. Line 44 on the other hand which includes the Cooley-Tukey algorithm loops each group, allowing multiple FFT operations to be computed concurrently which was beneficial for large input sizes, thus significantly reducing computation time. TimeFreq.cs received parallelisation changes from line 28 to line 67 with the STFT is computed for each window. Parallelism enables concurrent calculations of the FFT for different windows of data which significantly reduces the required time for the STFT, especially for longer signals. Normalisation step was also parallelised to distribute the workload of dividing each element by the max value. This was especially helpful for particularly large matrices.

# 9.0 Reflection and Conclusion

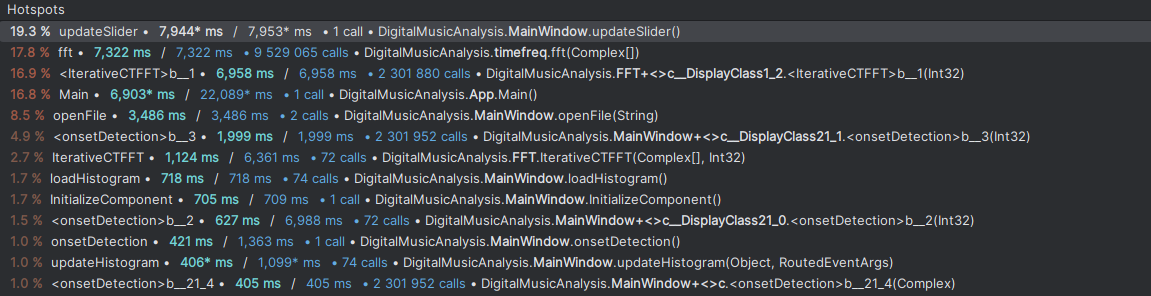
Throughout the parallelisation process, I gained a deeper understanding of the complexities involved in parallel computing such as load balancing, memory contention whilst ensuring data consistency. Valuable knowledge was gained seeing firsthand how various optimisations could impact performance. Overall the attempt at parallelising this task was quite successful, seeing performance improvements with reductions for FFT and onset detection. While the parallelisation was effective, there were still area for improvement. For instance, the program could’ve been optimised better with better load balancing techniques.

There were many things in hindsight that could’ve improved the application. The original plan for my task originally was to use GPU acceleration such as CUDA to measure the difference between sequential CPU processing and GPU acceleration, however this was beyond the scope of my knowledge and the use of CPU processing was used instead.

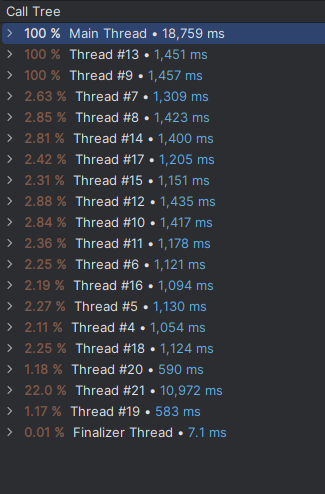
In conclusion, the parallelisation effort was both educational and rewarding. The application showed marked improvements in performance, and I learned a great deal about the details of parallel computing. While there were many challenges along the way, the final outcome was successful with the application being more efficient and scalable. There is always more to explore and optimise, but given the constraints and the tools at hand, I believe the outcome was as successful as it could have been.

# Appendix

#### Appendix 1: Parallelised thread hotspot



#### Appendix 2: Thread hotspot compressed



#### Appendix 3: Sequential hotspots

A screenshot of a computer program

Description automatically generated

#### Appendix 4: Sequential hotspot compressed

A screen shot of a thread

Description automatically generated

#### Appendix 5: Flame graph sequential taken from dotTrace

A screenshot of a computer

Description automatically generated

#### Appendix 6: Flame graph parallelised taken from dotTrace

A screenshot of a computer

Description automatically generated

#### Appendix 7: Sequential Spectrogram

A screenshot of a computer

Description automatically generated

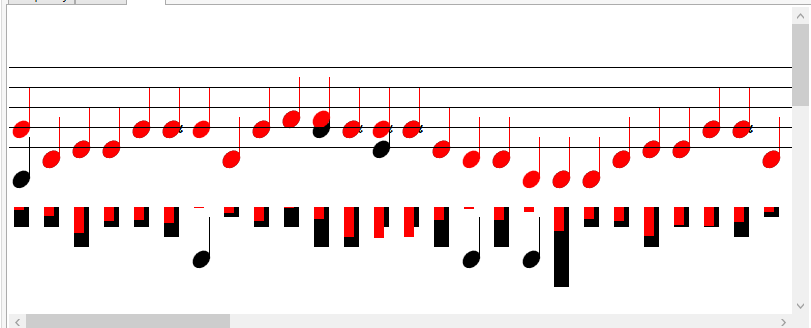
#### Appendix 8:Parallelised Spectrogram

#### Appendix 9: Sequential musical notes

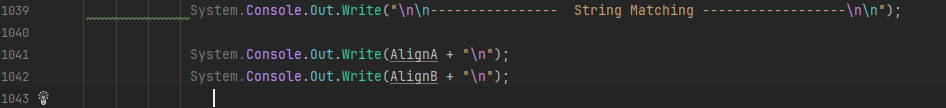
A screenshot of a music note

Description automatically generated

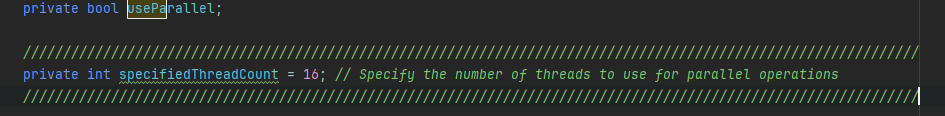
#### Appendix 10: Parallelised musical notes



#### Appendix 11:



#### Appendix 12







#### Appendix 13

A computer screen shot of a program code

Description automatically generated

#### Appendix 14

A screenshot of a computer program

Description automatically generated

#### Appendix 15

A screen shot of a computer program

Description automatically generated

#### Appendix 16

A screen shot of a computer program

Description automatically generated

#### Appendix 17

A screen shot of a computer

Description automatically generated

#### Appendix 18

A computer screen shot of a program code

Description automatically generated