Parallel Video Editing

Group 23: Cameron Bish, Rain Qin, Sravana Kumar Reddy
Departments of Electrical and Computer Engineering
University of Auckland

1 Introduction

The problem that we were faced with was to design a parallel video editing application. The focus of our application is to improve usability and performance of the design. This means that we had to apply different parallelisation concepts to enhance the performance of video filtering. This includes more than just concurrency concepts as the aim of our project is to increase the speed up of filtering an input video. We have developed two implementations, sequential and parallelised versions and tested the performance of each.

2 Literature Review

This book [1] talks about the design patterns of a parallel programming system and the issues faced during computational tasks and how one can overcome these issues efficiently. Section 1.2 discusses about the performance issues while performing computational tasks. The potential for scaling performance of a parallel system is constrained by the algorithm's span i.e. the time it takes to perform longest chain of sequential tasks.

When designing a parallel algorithm, there are important aspects to focus on. These include the total amount of computational work, the span which is the time taken to complete these tasks, and the total amount of communication including shared memory. These aspects apply to us especially the last one. As we are focusing on editing video files on the same system, file input/output can become a bottleneck to the system. This relates to the fact that only one file can be read from disk the system at a time, and it would be inefficient to store whole videos into memory. The book [1] also focuses on a shared memory machine model in which all parts of the application have access to the same shared memory address space.

3 Methodology

The main aim of our application is to apply parallelisation concepts to apply filters to videos. Since it is a desktop application with a graphical user interface we have applied concurrency to ensure background performance does not affect its responsiveness. Our graphical user interface has been developed using Swing. However, Swing alone does not have the capability to play media files. Therefore, we used JavaFX to input a media player into our Swing GUI. We have applied parallelisation concepts and allow the user to choose between parallel or sequential processing of a video. The main goal of our application is to focus on usability and performance. The main aspect we have considered is performance as we have applied many techniques to improve parallelism to enhance performance. The user interface of our design can be seen in Figure 1.

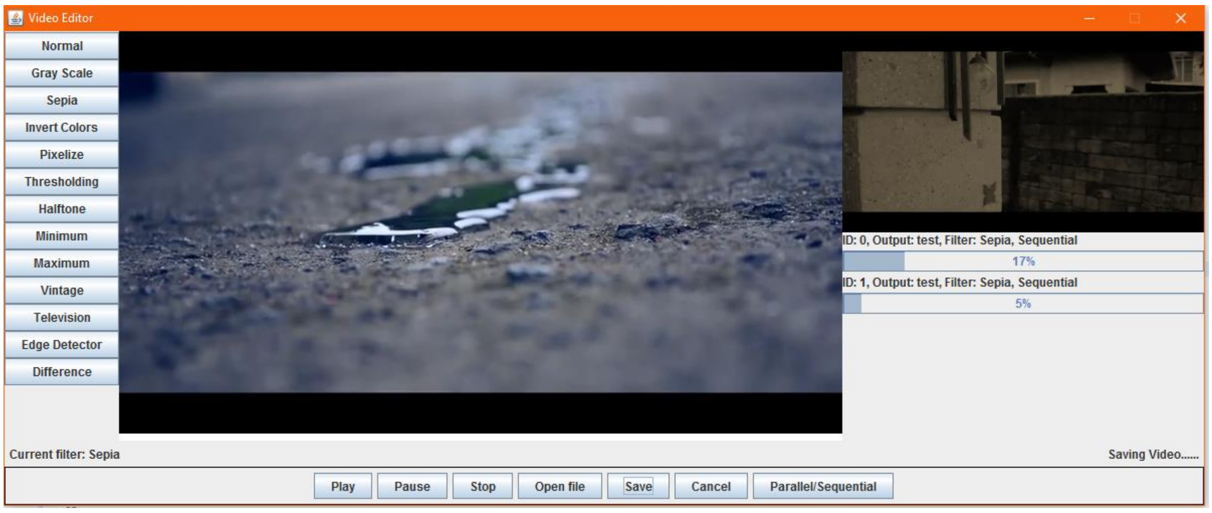
4 Libraries

In our implementation, we have used multiple libraries such as JavaCV, FFmpeg and JavaFX. These libraries help in providing the necessary tools to modify and display video files as needed for our application.

4.1 JavaCV

JavaCV is a Java wrapper which incorporate libraries such as OpenCV and FFmpeg. This means that it takes the functionality of these API and implements them in classes so that they can be used in Java development. It simplifies the use of the underlying libraries by simplifying interfaces.

In our implementation, we have used JavaCV for all video related processing for e.g. sequential filtering of a video file. There are generic classes used as general line, those are extended to create different classes based on the respective needs. More about these classes will be said in Section 5.1.



ID: 0, Output: test, Filter: Sepia, Sequential
17%
ID: 1, Output: test, Filter: Sepia, Sequential
5%
Current filter: Sepia
Saving Video......

Fig.1 Graphical user interface of our design.

4.2 FFmpeg

FFmpeg is an open source API that has libraries for handling multimedia data. Multimedia applications like VLC Player use FFmpeg. FFmpeg includes:
● libavcodec - an audio/video codec library used by several multimedia apps
● libavformat - an audio/video container mux and demux library
● the ffmpeg command line program for transcoding multimedia files.

In our implementation, we have used the FFmpeg API to perform, video splitting, video combining and audio mapping.

4.3 JavaFX

JavaFX is a set of graphics and media packages that enable developers to design rich client applications. JavaFX application code can reference API's from any java library. In our implementation, we have used JavaFX to provide a media player which is integrated into our Swing GUI.

5 Implementation

5.1 Sequential Video Processing

In JavaCV there are classes that aid with sequential filtering of a video file. The main classes that we used in our implementation were Frame, FFmpegFrameGrabber, FFmpegFrameFilter and FFmpegFrameRecorder. The Frame class is used to represent a frame video. It holds the image and audio samples for a frame in buffers. In our case, the image of a Frame can be accessed and modified as required. A frame grabber object is instantiated with an input video. It allows you to open the file and access the individual frames. It is used as an input stream for retrieving frames. Once you have received a frame you can modify it by pushing and pulling it to apply a preset FFmpeg filter to the image of the frame. The FFmpeg recorder is used to record frames to an output file. It is used as an output stream so that modified frames can be written to the output. The timestamp of a frame can be retrieved so that it can be recorded to the output file at the same timestamp.

We can use these classes to filter videos by grabbing a frame from a video, applying to filter to it and then recording it to the output video. This is continually done until there are no frames left in the video. After this process, it signals to the EDT thread that it is done, which removes the processor from the collection. This sequential video filtering process is described visually in Figure 2.

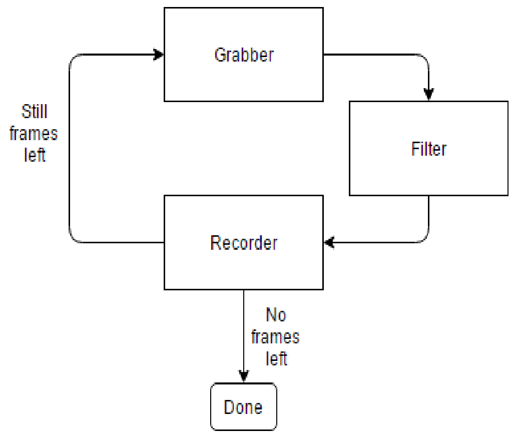


Fig.2 The sequential video filtering process.

The way in which we have done our implementation is that the user is able to process multiple videos sequentially at the same time. This is done by adding sequential processors to a collection every time

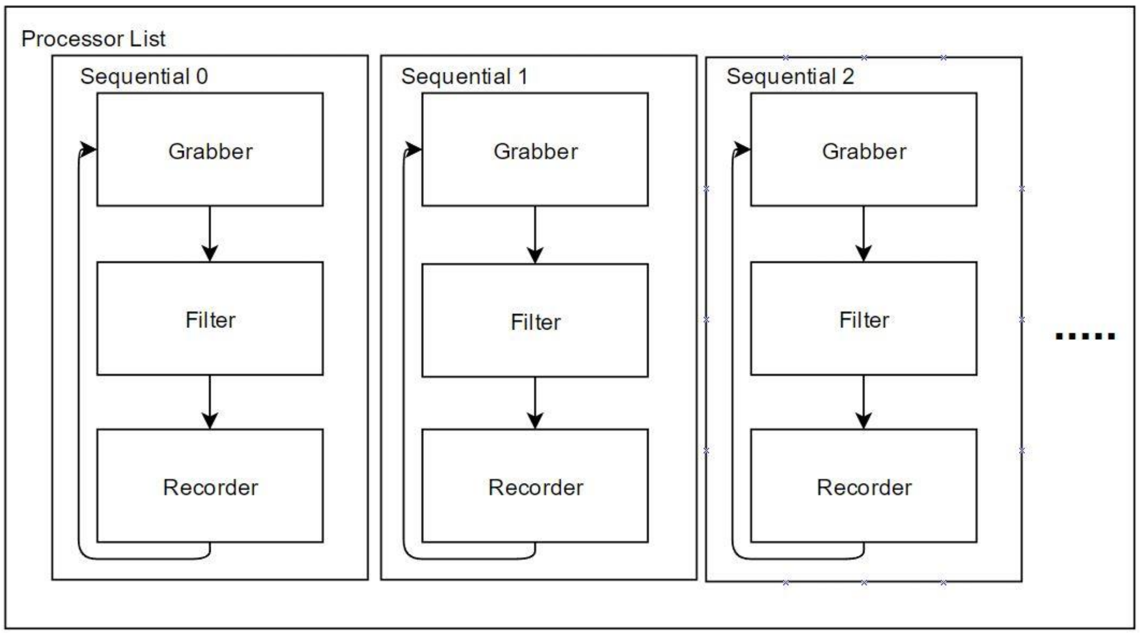


Fig.3 Multiple input videos being processed sequentially simultaneously.

the user wants to process a video in sequential mode. This is visually represented in the Figure 3. After each processor is finished its computation it removes itself from the collection.

5.2 Parallel Video Processing

The user is also able to process a video in parallel. This means that the video is split into sections. These sections are then filtered in parallel and combined to form the output video. At this stage, we have only allowed one parallel processing instances to exist at one time. This was decided as the parallel processing of a video uses a thread for each core of the systems processor that it is running on. For example, if there are four cores on the system's processor, the input video will be split into four parts. Each will be processed by an individual thread. If there were multiple parallel instances of processing, too many threads will be created.

Once the video has been split into sections, the sections of the video are filtered by using the same process as with sequential processing. The sections are treated as individual videos and frames are grabbed, filtered and recorded sequential in that section. After each processor is finished its computation it removes itself from the collection. A visual representation of our parallel video editing model can be seen in Figure 4.

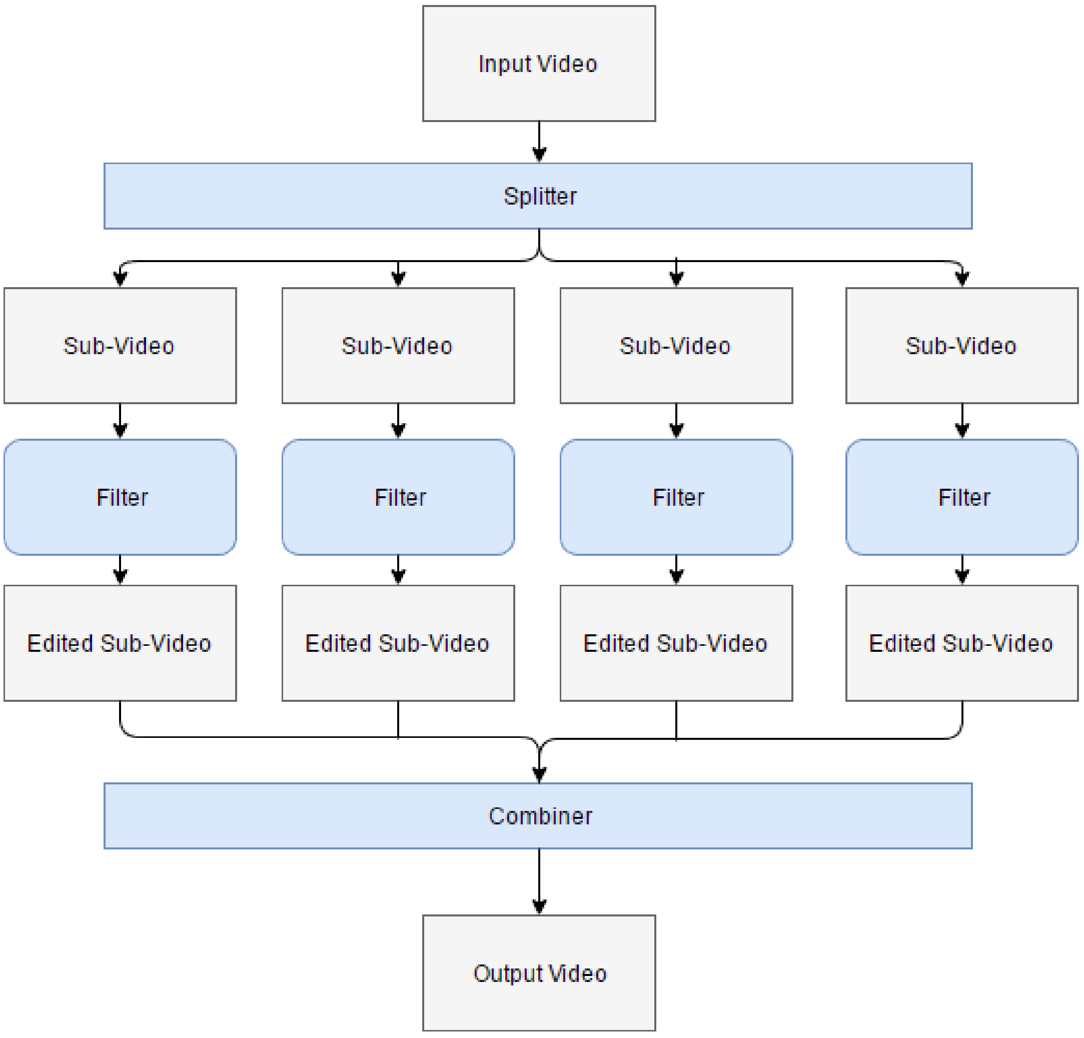


Fig.4 Overview of out parallel video processing model.

6 Parallel Processes

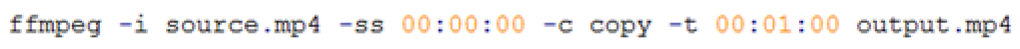
For editing a video in parallel, our group decided to apply ParaTask [1]. This was decided because it was designed for desktop applications which implement a user interface. ParaTask aims at taking some of the abstraction of parallel processing away from the programmer. It supports one-off tasks, multi-tasks and I/O tasks which are needed by our project. Moreover, it provides the ability to manage dependencies between tasks.

The main idea of our project is to parallelise video filtering by splitting the input video in several sub-videos. Then arranging a thread to pick a sub-video each for filtering. When the filtering process is finished in each thread, all the filtered sub-videos will be combined by a one-off task.
Essentially there are three major tasks of our parallel implementation:

1. Video Splitting
2. Video Filtering
3. Combining for Output

6.1 Video Splitting

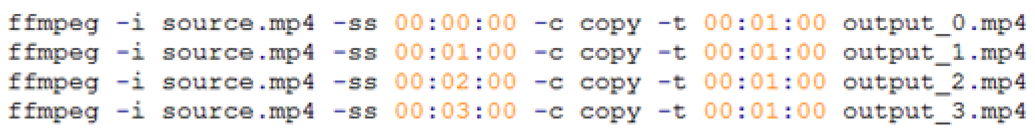
We tried to use pure java code to split the input video into sections. We concluded that it was not efficient enough for our application even for a short video (approximately 3 minutes). Fortunately, the API we have been using, FFmpeg, provides a function that enables us to split a video into sections through the command line. This method is indeed more efficient. The start and end times need to be determined therefore, firstly we get the video length and calculate the durations for each section that we will split the video into, to partition the input video using the FFmpeg command line.



```
ffmpeg -i source.mp4 -ss 00:00:00 -c copy -t 00:01:00 output.mp4
```

Fig.5 FFmpeg command for splitting video.

For example, for a four-minute video, if we want to split it into four parts, the duration will be 00:01:00 minute for each part. Thus, the time range will be [00:00:00, 00:01:00], [00:01:00, 00:02:00], [00:02:00, 00:03:00], [00:03:00, 00:04:00]. However, the second time in the command is not the end time. It is the duration. To split one video into multiple sections, if the available processor has four cores, then we need to run four commands. The commands are shown below:



```
ffmpeg -i source.mp4 -ss 00:00:00 -c copy -t 00:01:00 output_0.mp4
ffmpeg -i source.mp4 -ss 00:01:00 -c copy -t 00:01:00 output_1.mp4
ffmpeg -i source.mp4 -ss 00:02:00 -c copy -t 00:01:00 output_2.mp4
ffmpeg -i source.mp4 -ss 00:03:00 -c copy -t 00:01:00 output_3.mp4
```

Fig.6 FFmpeg command for splitting an input video (into four sections in this example).

The initial plan was to split the video using a one-off task. Since we have multiple commands to execute, we can utilise multi-tasks by applying ParaTask. We had an experiment for this task of splitting the input video into multiple sections. As it can be seen in table 2 the multi-task implementation provided us with the best results.

	Sequential	One-off (TASK)	Multi Tasks (TASK(4))

Time Spent	4139ms	223ms	73ms

Table.1 Splitting Task Time.

6.2 Video Filtering

After splitting the input video, we need to add the chosen filter to each sub-video in parallel. To achieve this, we let a thread pick a sub-video and do addFilter function (add filter task). The number of tasks depends on the number of sub-videos.

The basic process of adding filter is that the filter will be added to every frame of the sub-video and then the frame is written to the specified output file. Adding filter to each video is a sequential process because there are dependencies between frames. We could change this sequential process to parallel by splitting the video into frames so each frame will be independent. But it will cause too much overhead especially for long length videos therefore it would not be worthwhile.

6.3 Combine for Output

Combining video is not straightforward. The sub-videos are in MP4 format which cannot be combined directly. The MP4 videos need to be converted into ‘ts’ files first then combine these ts files and set the output file as MP4 format. The FFmpeg API provides the function to combine videos through its command line which can be applied into ProcessBuilder.

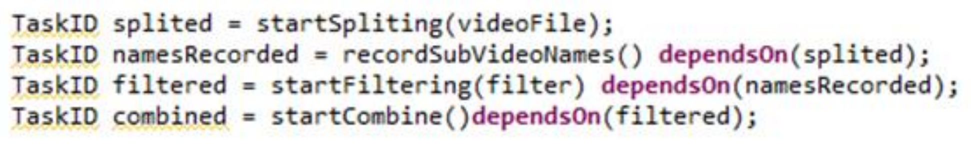
7 Parallelization Concepts

In our project, there are many computational tasks including splitting the input video, filtering sub-videos and combining them for parallelization process. We planned to make splitting video as one-off task but when implementing it, we found FFmpeg provides a way which supports us to change this task to be multi-tasks which can be applied by multi threads through using ParaTask. The filtering task can be executed in parallel because it works on several sub-videos. The final task, combining would be a one-off task.

For all I/O task we have used the Event Dispatch Thread (EDT). When video processing takes place, this is done using a SwingWorker instance. This allows us to achieve concurrency, meaning we can still use the responsive GUI whilst video processing is done in the background. This is an important concept when creating any desktop application with a GUI.

In our parallel implementation, we applied static scheduling. This is because the size of sub-videos is equal, as we are splitting them into equal parts ourselves. The time that each thread spends on each sub-video would be close enough to the same that it would not affect the performance of our application. Therefore, we determined that other scheduling policies such as different dynamic scheduling type would not really be suitable for our project.

In our parallel implementation, there are dependencies existing between these tasks (splitting, filtering and combining). Only once the splitting processed has finished, the splinted videos can start their



```
TaskID splited = startSpliting(videoFile);
TaskID namesRecorded = recordSubVideoNames() dependsOn(splited);
TaskID filtered = startFiltering(filter) dependsOn(namesRecorded);
TaskID combined = startCombine()dependsOn(filtered);
```

Fig.7 Dependency management of our ParaTask implementation.

filtering tasks and if every frame of these sub videos are not filtered, it cannot combine the filtered video together. ParaTask makes it easy to manage the dependencies between tasks.

During our implementation of ParaTask, we found an issue. The issue was that if we only used the "dependsOn" method to manage dependencies between tasks, some filtered videos would not be ready when the program reached the combine task. For example, a video is split into four sub videos and these four videos picked by threads to start adding filters. The filtering task started but while not all filtered sub videos generating completed, combine task might start. This may cause, only some (not all) of filtered sub videos entered combine task so that the combined video is not completed. After checking ParaTask documentation, we found that the "waitTillFinished" method solved this issue.

8 Results

For our design, we wanted to perform some experiments to see if and how much speed up there is between our parallel and sequential implementation. To make it fair these tests were done on the same machine and there was only one video being processed at a time for all scenarios.

To fully compare our two implementations, we thought it would be best to compare different length videos. This is because our hypothesis was that the longer the video, the greater speed up we will achieve. This assumption was formed by the fact that with a short video, it would not be as efficient to split the video into sections, filter it, and then combine it as it would be for a longer video. This is because the sequential implementation is already efficient enough without add extra overhead.
Table 2 below shows the results from different ways for video editing. As we can see, a 1-minute takes approximately 11924 ms to finish filtering in sequential and similarly, the same video used 10265 ms to complete this job in parallel. There was no big difference between this two method when dealing with small video. For another experiment, when we edit a 3-minute video, the parallel version was about 25000 ms faster than sequential version to finish the whole tasks. For a 30-minute video, compared with our parallel implementation, the sequential method performed 30 seconds slower.

Video Length	1 minute	3 minutes	30 minutes
Sequential Time	11924 ms	152761 ms	231989 ms
Parallel Time	10265 ms	127207 ms	200683 ms

Table.2. Time cost - Video editor sequential implementation vs parallel implementation.

9 Design Limitations

JavaFX is only compatible with certain video formats. As we are using JavaFX to display video in our GUI, only some file types can be opened. The FFmpeg framework that we have used for editing our videos can work with nearly any video format. In future work, we would look to another media player that can extend our platform of play more formats. This limitation meant that we were mostly working with '.mp4' video files.

JavaCV provides a good way to filter videos. However, we found issues with audio not being written to the output file properly. This was a major issue, as there is no point in filtering a video if it degrades the video quality by distorting the sound. This issue was fixed by only capture the image of the input video and then using the FFmpeg API to map the audio from the input video to the output at the end. Also with our JavaCV implementation we found that some of the FFmpeg filters were not compatible. This restricted us on the types of filters we could use in our design. Looking back, we feel as though it would have been better to only use the FFmpeg API for all video editing, and not use JavaCV.

We experienced a limitation with using the FFmpeg API. When we split the input video by using command line provided by this API, we found that there is sometimes a problem with losing some frames. On some occasions, this causes a stutter in the output video. In addition, sometimes when we filter the sub-videos and combine them, the final output video may lose some frames. This influences our final filtered video quality for the parallel implementation. In future work would look deeper into this API to find a solution. If fixing this issue is not possible because of the API's limitation, we might consider changing API to provide better output quality.

10 Conclusions

Overall, we see that our parallel implementation provides good speed up compared to the sequential version. We have observed that it works better on longer videos. Due to the additional overhead created in the form of splitting and combining, it does not work as efficiently with shorter videos. With our design, we found some limitations due to our choices in libraries used in the project. A lot has been learnt about video editing and we will take our observations into any future work we carry out around video editing.

References

[1] M. McCool, A. Robison and J. Reinders, Structured parallel programming, 1st ed. Amsterdam: Elsevier/Morgan Kaufmann, 2012.
[2] "Parallel IT - Parallel Iterator, Parallel Task and Pyjama", Parallel.auckland.ac.nz, 2017. [Online]. Available: http://parallel.auckland.ac.nz/ParallelIT/PT_About.html.

Contributions

Task	**Contributor/s**
GUI design	Cameron Bish, Sravana Reddy
GUI media player integration	Rain Qin, Sravana Reddy
GUI concurrency	Cameron Bish
Sequential video implementation	Cameron Bish
Parallel video implementation	Rain Qin
GUI integration of parallel/sequential implementations	Cameron Bish
Report	Rain Qin, Sravana Reddy, Cameron Bish