SOFTENG 751 PROJECT REPORT

Hanliang Ding
Jerry Fan
Aneesh Thunga
Department of Electrical and Computer Engineering
University of Auckland
https://github.com/Nemeritz/A1_SE751_G27/releases

1 INTRODUCTION

Current parallel programming techniques are separated into two different categories, language-based approach and library-based approach. Language-based approaches tend to be more intuitive for the programmer as it wraps all the parallelization into language constructs, so the parallel code is very similar to the sequential code. However, this approach falls short when it comes to allowing the programmer to customize things such as granularity of tasks. Another problem of language-based approaches is that it is often difficult to get coding assistance from conventional IDEs such as IntelliJ, Eclipse and Visual Studio.

Library-based approaches on the other hand allows the programmer to use APIs to control many aspects of the parallelization of the program. However, this is often hard to put together for the programmer as assembling these functions into a working program is always difficult.

@PT is a programming construct which builds on its predecessor ParaTask and utilizes annotations in java to make the programmer's job of parallelization easier while still allowing the programmer to perform fine grained controls over the parallelization [1].

This report will first report on other related works which could be possible substitutes for @PT. Then we will describe how our implementation of the GUI application utilized @PT to achieve speed up, increase GUI responsiveness and reduce programming complexity. The results will then be evaluated and discussed in the last section.

2 RELATED WORKS

2.1 SWINGWORKER

Swing utilities allows the GUI thread also known as the Event Dispatch Thread (EDT) to be separated from other computationally intensive background tasks. The background tasks can utilize invokeLater() and invokeAndWait() to place GUI update events into the EDT queue so that the GUI is responsive. However, with SwingWorkers the programmer has to manage all the dependencies between the tasks and have to manually separate out the tasks that are to be ran on the EDT and other computational threads. This makes it hard for the programmer to restructure the code if needed and all the responsibilities of parallelization falls on the programmer.

2.2 PYJAMA

Pyjama provides java support for parallelization following the OpenMP approach and it incorporates the shared memory fork and join model from OpenMP [2]. Much like @PT, Pyjama also provides GUI support for responsive GUIs in the form of #gui directives. Pyjama solves the problem of EDT being used as the master thread in the fork and join model by creating a substitute thread as the master thread so the EDT can go back and allow the GUI to be responsive again. However, unlike @PT's GUI annotations the programmer has to worry about which parts of the code might need to be GUI aware and add directives for that as well as for what type of GUI operation it is.

In terms of effort required to produce parallel programs in Java, Pyjama proved to require less effort than @PT [1]. However, @PT promotes object-oriented thinking whereas directives from Pyjama is quite lacking in this regard.

Pyjama also allows developers who come from an OpenMP background be able to quickly get used to its directives and the fork and join model while being in a higher-level language like Java. Pyjama has showed promise but is still in the research and development stage as many future features have not been implemented yet.

2.3 JOMP

Jomp is very much like Pyjama in the sense that it is a directive-based approach which tries to follow the OpenMP models [3]. Unlike Pyjama, Jomp does not have any GUI specific directives which would make GUI updates a lot more difficult to implement. Jomp however does have more support in terms of OpenMP directives than Pyjama is overall more further into the development stage. That being said, Jomp still has some problems such as using directives for code which has been annotated as it can sometimes fail to parse the source code [4].

3 GUI APPLICATION DESIGN

3.1 INTENTIONS

The application that we made is a face gallery application. This application reads in pictures from the path as specified by the FACEGALLERY_DATASET environmental variable and displays them in the application. Any pictures that does not contain faces will be displayed with a blur and darken effect. Progress bars are available to show the progress of each task and the responsiveness of the UI and table of execution times for each task is shown in seconds.

The reason for choosing a face gallery application is that it can effectively utilize @PT to perform performance enhancements in terms of speedup and responsiveness. The tasks can also be decomposed as shown in Figure 1 effectively to allow pipelining and parallelization. The communication to the Google Cloud API allows for effective use of IO tasks in @PT to achieve speedups.

3.2 RUN MODES

The application has four run modes; sequential, concurrent, parallel and pipeline. The following section will discuss the advantages and disadvantages of each run mode implementation.

3.2.1 Sequential

In sequential mode, each module and their operations are processed sequentially. This means the GUI can only update in between each module i.e. when file reading finishes or face detection finishes. This is not responsive and has a very long execution time. The parallel mode on the other hand has a responsive GUI and much less execution time with the help of IO tasks.

The blocking of each task can be seen in, each task has to finish execution before the next task can start and UI updates are only done at the end of the execution as a finalization update. The only advantage of sequential is that the code is relatively simple as the programmer doesn't need to worry about parallelization and concurrency related issues.

3.2.2 Concurrent

Concurrent mode in the project was implemented with all tasks annotated with @Future. This means These four tasks can be pre-empted, so GUI updates can be done during the execution of these tasks. However, this is all useless for the GUI if it doesn't know when to access the images for displaying. To solve this problem a blocking queue is used to help synchronize the GUI's reading of the images and all the other tasks' writing to the images.

The blocking queue records the index of the element which is ready to be read by the GUI. The intermittent GUI updates can increase execution time, but it does improve the responsiveness of the GUI.

3.2.3 Parallel

For the parallel implementation we used Multi IO tasks for file reading and IO tasks for face detection. This means that the wait time for IO intensive tasks such as face detection are greatly reduced because a new thread is created for each image. The execution time for the face detection task is greatly reduced due to the creation of many threads as IO tasks are implemented with cachedThreadPools.

Parallel execution also means that the GUI will not be blocked just like the concurrent execution. This is done with a BlockingQueue which is used to indicate which images have finished processing.

3.2.4 Pipeline

The pipelined implementation means that each task's individual workers are done in a pipeline fashion. This is done with a blocking queue to ensure that the depending tasks for the corresponding image element is completed. The inner loop worker then performs the actual execution of the task.

The pipelined mode runs at a similar rate to the parallel mode but is a lot more responsive than the parallel mode there are no inter task dependencies but only inter worker dependencies.

3.3 TASKS

3.3.1 Task Decomposition

Four tasks have been implemented as part of this application. These are file reading, thumbnail generation, face detection and blur & darken. A decomposition of the dependencies of these tasks can be seen Figure 1, where the processing occurs from the leftmost task and progresses towards the right. This application leverages Google Cloud Vision to perform face detection tasks, which is a public external network service hosted on the Internet. In an attempt to create a more object-oriented task setup, each

task is set up as its own class with all actions relating to that task written as methods. Whilst this may seem irrelevant, there were some issues relating to @PT limitations when coding with this abstraction that are discussed in 5.1.3.

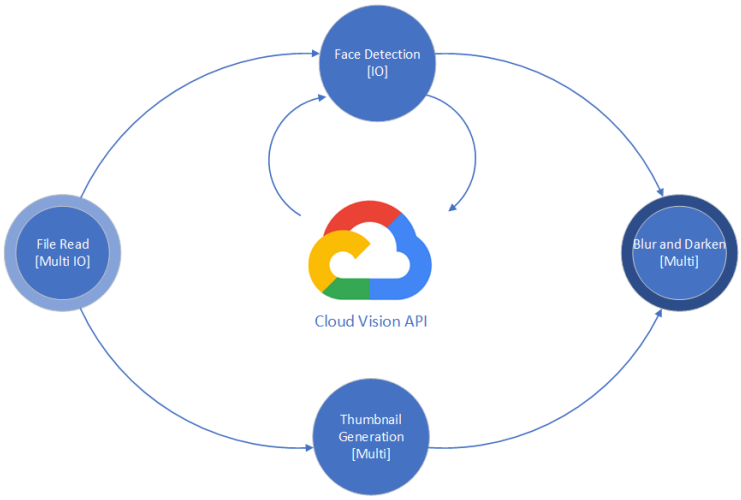

Figure 1. Task Decomposition

3.3.2 File Read

The file read task involves getting data from the local disk that contains the image dataset to be processed. File enumeration is done on the creation of a file reader object as it takes negligible time to list files and get pointers to the files, so the process was excluded from parallelization. The number of files were important to know before task execution so that the appropriate data structures can be set up for the various tasks and the GUI interface. Only JPEG, PNG and GIF files are considered as images from the dataset folder. When run, the file read task attempts to read all images as byte arrays and stores it in memory. This is implemented as a multi-interactive task in the parallel run mode.

3.3.3 Thumbnail Generation

This task involves resizing the byte arrays incrementally to produce a high-quality thumbnail of the full image represented by the byte array, using the built in BufferedImage class and associated filters. This is implemented as a multi-one-off task in the parallel run mode.

3.3.4 Face Detection

This task uses the full image from the byte array and sends it to Google's Cloud Vision API so that a face detection algorithm can be used to detect if the image contains faces. The detections once returned are turned into Boolean true or false values. In sequential or concurrent mode, this task has an optional behavior to send images in batches of up to 16 images per request. This batch mode allows face detection to be done with less network requests, so that the effect of network latency can be reduced. This is implemented as an interactive task in the parallel run mode.

3.3.5 Blur and Darken

This task uses results from the two previous tasks, performing two image filtering operations provided natively by Java and a third-party library to darken and blur the images respectively if face detection for the image returns false. This is implemented as a multi-one-off task in the parallel run mode.

3.4 USE OF @PT CONSTRUCTS

3.4.1 @Future

This is used extensively in all tasks, as it provides the basic functionality of starting a new thread running a method. The one-off futures were mainly used to create threads running in parallel to the Swing EDT that awaited on tasks to complete, or to unblock a thread should waiting be required. It was also useful to use @Future's dependency declarations to create execution barriers in unsynchronized async methods (which many methods in the task classes are) so that interim GUI updates can be performed at the right times, although they come at a cost of an extra thread used purely for synchronization. This was considered to be a reasonable trade-off in performance for code clarity. The multi-task type, both interactive and one-off were used by the majority of tasks. Whilst these tasks could also be implemented through just pure interactive and one-off tasks in future groups, it was found that the multi-task type could be used to limit the amount of threads spawned during execution, to prevent the application's worker threads from consuming too many system resources and freezing the host machine. It was found to be useful for processor intensive tasks such as thumbnail generation, and for disk intensive tasks such as the file reading.

3.4.2 @Gui

This construct was used intermittently at synchronization points along the task execution, so that the Swing GUI could update with the latest processed images and task progress.

4 BENCHMARKING

4.1 SETUP

The performance of the application was measured by running it on a virtual machine. The virtual machine was configured to run with 1, 2, 4 and 8 processors. An execution cap of 80% was put in place as the host machine's idle CPU use was between 0% - 15%. The host machine has the equivalent of 4 processors (2 hyperthreaded). This was done to compare the execution times with different number of processor cores. The guest operating system chosen was Xubuntu 18.04 for our benchmarking. The application was modified to run with 32 threads maximum so that the processors could be saturated.

4.2 METHODOLOGY

The time taken for all the implemented tasks were compared with different run modes. All tasks were run in sequential, concurrent, parallel and in pipeline modes. The benchmarking was performed on a dataset of 32 images, out of which 16 of them contained faces. Each result was averaged out of 50 trials.

The program was run 10 times at the start to mitigate JVM cold start affecting the overall results of the later trials. All background tasks that could be stopped were killed to ensure all work is done on the application tests.

4.3 RESULTS

Below are some of the benchmarked results. Lower values are desirable. Pipelined time not measurable for individual pipeline stages within our timing framework and are not shown.

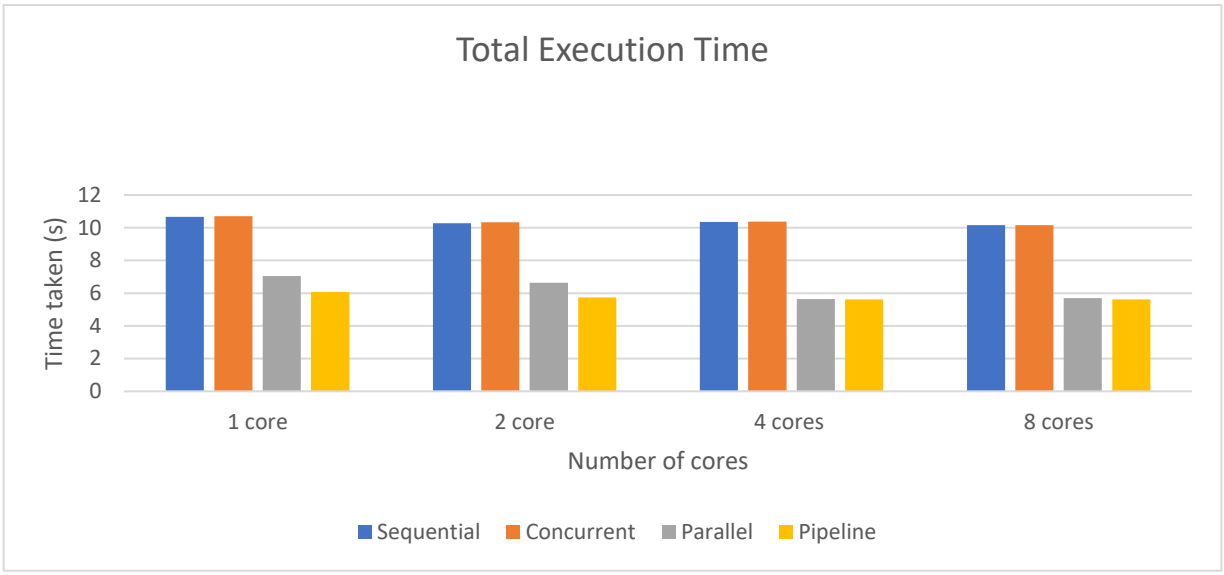

Figure 1. Time taken for total execution

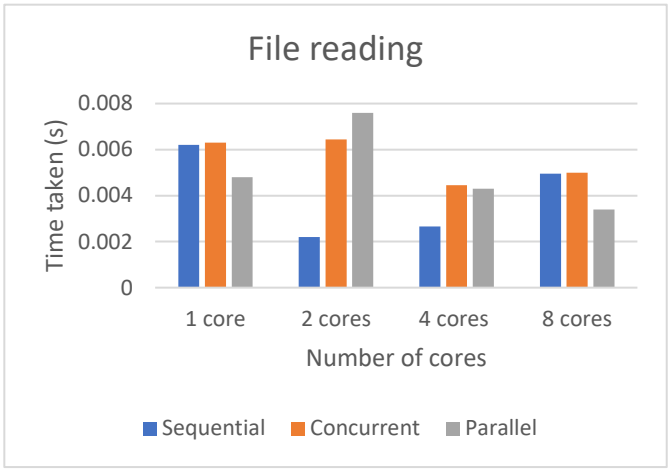

Figure 3. Time taken for file reading task

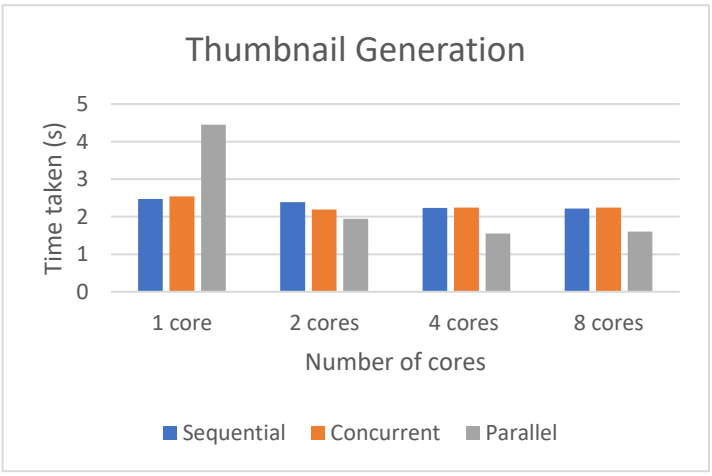

Figure 4. Time taken Thumbnail generation task

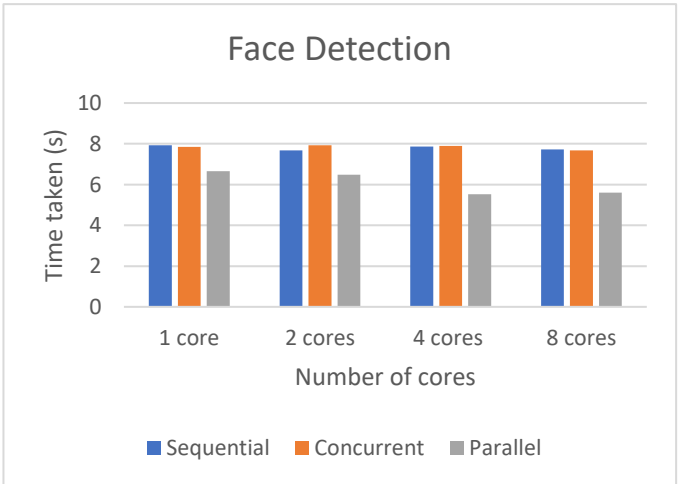

Figure 4. Time taken for face detection task

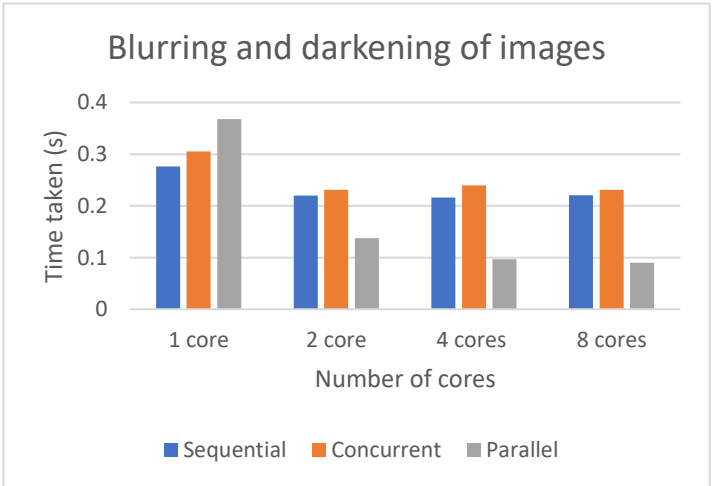

Figure 5. Time taken for thumbnail blurring and darkening task

5 DISCUSSION

5.1 LIMITATIONS OF @PT

5.1.1 Build Setup

One of the major difficulties in the use of @PT in the development of this GUI application was the build setup. Without using Java's built in annotation processor, a double compile process needed to be set up with the annotation processor and the Java compiler working to translate @PT to Java files using the PT runtime, and then to class files. The problems with this approach are that the interim objects generated during the annotation processing are not re-used if a source file is changed, but rather the whole process is restarted from scratch each time. Compared to normal java compilation that does not regenerate unchanged classes, this makes the compilation take a long time to complete. Debugging is also difficult as the final source code does not exist before compilation, and are different to the initial source code, requiring the programmer to understand the PT Runtime syntax generated which defeats the purpose of learning the easier to use annotations.

5.1.2 Java Annotations

Java annotations only apply to object declarations, methods and classes. This makes the @PT annotations somewhat awkward to use in practice, requiring many objects to be created if there are unrelated tasks with different return types. The requirement that functions return an object also means that wrapper functions have to be created to run any functions that return the 'void' type.

5.1.3 Programming Abstractions

As mentioned previously in the tasks section, an object-oriented programming style was applied to the design of the application. Following good practices, the task classes were written in a way that do not require any knowledge of how their return values are used, with this detail being known to the display controller (and the Tasker helper class) only. Whilst using @PT did not present a problem when interactions between future and normal objects were limited to within a method scope, they presented a large problem when moving between methods in different objects, as it was impossible to return a future object without the annotation processor inserting a synchronization point. This made it hard to enforce a single purpose policy to classes if the automatic synchronization is used. A possible solution to this problem could be to give programmers control over when future objects turn back into regular Java objects, through the addition of another construct @await, that inserts the synchronization point at the point where it is declared. The current implementation uses a large number of BlockingQueue objects in 'await' threads, one required for each set of tasks launched to provide intermittent GUI updates.

5.1.4 Bugs

This is explained in more detail in the bug report filings available at:
https://github.com/Nemeritz/A1_SE751_G27/issues

5.2 GUI IMPROVEMENTS

An issue noticed after implementation of the GUI was that the images do not update fast enough to provide a 'responsive' view of what the tasks are doing. This difference is clearly seen if the statistics are compared against the images, with statistics updating frequently. This is due to the relatively large

amount of icons being created as JLabel objects in each update cycle. A future improvement could be to use a canvas and render images in memory directly on to the canvas instead.

6 CONCLUSIONS

From the results, a fairly noticeable performance increase could be seen during benchmarking of the tasks from the sequential and concurrent to the parallel and pipeline run modes when the VM was assigned more than one processor. When only one is assigned however there seems to be an increase in time taken for CPU intensive operations, but still a decrease in time taken for IO tasks such as file access or network requests. This makes a good case that @PT is overall useful in getting a performance boost in applications easily on systems with multiple processors available, and IO tasks in the cases where only one processor is available. However, pure performance is not all that is important in GUI applications, the responsiveness of the GUI in the concurrent, parallel, and pipeline run modes created a far better user experience than that of the sequential run mode. As the concurrent run mode was hardly modified from the sequential, a good argument could be made that by just adding a few @Future annotations, a Swing GUI application could be made responsive. However, it should be noted that using @PT does come with difficulties as it complicates the build process and introduces PT runtime code that is automatically injected into the source file, which can be either good if it produces behavior that is desired, or bad if it does not. The decreased control does pose a problem in object-oriented programming but can be resolved by simply using a custom synchronization method.

7 CONTRIBUTIONS

Hanliang Ding	Report, Presentation, Debugging, File IO, GUI Implementation
Jerry Fan	Report, Presentation, Face Detection, Image Blur & Darken, Design
Aneesh Thunga	Report, Presentation, GUI Implementation, Testing, Benchmark

8 REFERENCES

[1] Mehrabi, M., Giacaman, N., & Sinnen, O. (2018). @PT: Unobtrusive Parallel Programming with Java Annotations.

[2] V., Giacaman, N., & Sinnen, O. (2013). Pyjama. *Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores - PMAM 13*. doi:10.1145/2442992.2442997

[3] Bull, J. M., & Kambites, M. E. (2000). JOMP---an OpenMP-like interface for Java. *Proceedings of the ACM 2000 Conference on Java Grande - JAVA 00*. doi:10.1145/337449.337466

[4] Belohlavek, P., Steinhauser, A. (2015). OpenMP for Java. *https://is.cuni.cz/webapps/zzp/download/130155774.*