Project 9 - Java GUI app using @PT

Report by Group 14 - Andy Tang, Joseph Dumogho, Lite Kim

Table of Contributions

Task	Andy	Joseph	Lite
Research	33%	33%	33%
Implementation	33%	33%	33%
Report	33%	33%	33%
Presentation	33%	33%	33%

References

[1] M. Mehrabi, N. Giacaman and O. Sinnen, "Annotation-based Parallelization of Java Code," in *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, Lake Buena Vista, FL, 2017.

[2] M. Mehrabi, N. Giacaman and O. Sinnen, "Unobtrusive Asynchronous Exception Handling with Standard Java Try/Catch Blocks," in *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, Vancouver, British Columbia, 2018.

[3] M. Mehrabi, N. Giacaman and O. Sinnen, "Unobtrusive Support for Asynchronous GUI Operations with Java Annotations," in *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, Vancouver, British Columbia, 2018.

Research

Within the development of high-performance programs, developers look into utilizing extended libraries or language construct frameworks for parallelization.

The utilization of extended libraries provides portability, allowing the implementation of parallelism to be widely extended between different programs. However, utilization of libraries require heavy restructuring of the original sequential code and the implementation may be inconsistent due to the case where different individuals follow different design methodologies. This results in programs with code that is difficult to understand and difficult to modify.

Language construct frameworks provide a means to simplify code by hiding the complexities of parallelization and promotes the avoidance of writing boilerplate code. Language constructs however, do not provide the portability that extended libraries do as different programs satisfy different compilation requirements. [1]

Annotated Parallel Task (@PT)

Annotated Parallel Task (@PT) is a language construct framework which utilizes java annotations, developed with the motivation of retaining the pros and improving upon the cons of the standard parallelizing concepts. This provides potential to parallelize code that is (close to) uniform, minimizes the restructuring of sequential code, and is portable.

@Future

To declare a task to run in parallel, @Future should be called at the invocation of the task. A task can be identified as either a one-off task (task with a single unit of computation), IO task (involves communications with input/output devices), or a multi-task (task where a single instruction is applied to multiple data sets)

By default, @Future declares the task to be one-off. To state otherwise, taskType should be defined as such.

```
      @Future(taskType = <<task type>>)
      Void task = foo();
```

If the type of task is defined as a multi-task, a taskCount can be defined with the value set as the number of subtasks within the task.

```
      @Future(taskType = TaskInfoType.MULTI, taskCount = <<value>>)
      Void task = foo();
```

Some tasks may be dependent on the output of other tasks which becomes a problem when declaring an annotation. Dependencies must be addressed so that parallelised outputs are identical to the sequential outputs. The depends keyword allows for explicit declaration of a dependency. Consider the following code. task3 is dependent on task1 and task2, so will not run until both tasks have completed.

```
      @Future(depends = "task1", "task2")
      Void task3 = foo(task2);
```

@PT can automatically identify and define dependencies of a task if the respective annotated task uses the return value of another annotated task as an input argument. This allows the user to write the following code which provides an identical result with the above.

```
@Future(depends = "task1")
Void task3 = foo(task2);
```

To process a task/s immediately after the completion of a parallelized task, the respective tasks should be defined with notifies.

```
@Future(notifies = "foo2()", "foo3()")
Void task = foo1();
```

@AsyncCatch

Consider a program that calls on a method that spawns a new thread. In this new thread, processes take place until an exception is caught. The exception needs to backtrack and find the exception handler method. If the handler was located at the method that spawned the new thread, the exception cannot be handled as the spawn of a new thread resulted a disconnection. @PT provides annotations that allows the user to explicitly define potential exception classes under throwables and allocate the appropriate handlers [2]

```
@Future(taskType=IO)
@AsyncCatch (throwables = {IOException.class}, {handlers =
"handler()"})
Void task = foo();
```

@InitParaTask

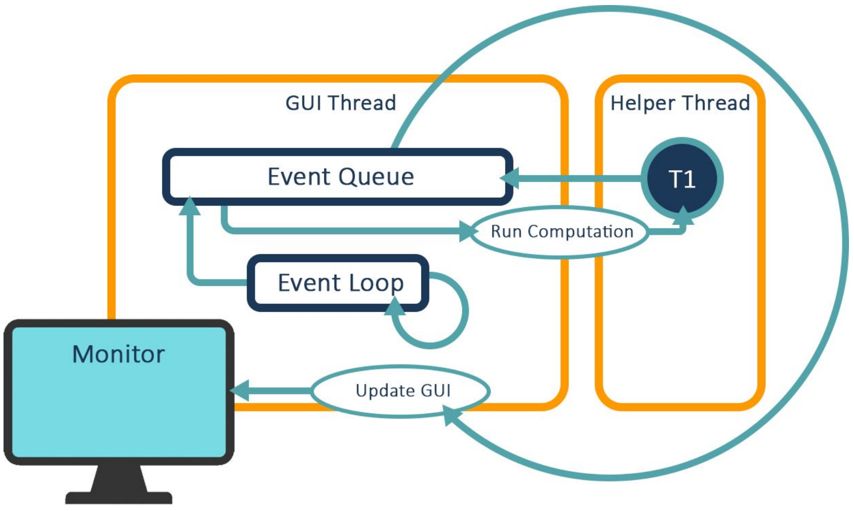
Called to configure the @PT runtime environment

GUI Parallelization

Development of GUI systems should always focus on keeping the GUI responsive and interactive. This can be achieved by separating the computational threads from the GUI updating threads allowing for computations to run in the background while the GUI always checks for interactive events.

GUIs are not thread safe. At the end of a computation, the computation thread should not directly update the GUI. Rather a single thread should run and update the GUI elements. This provides the following implementation:

A GUI thread always runs, checking for an interaction through the Event Loop. Once an interaction is detected, the GUI queues the corresponding task onto the Event Queue. The event queue sends the task to a helper thread to run the computations in the background while the GUI thread still continues to check for interactions. At the end of the computational task, the task sends a request to the GUI thread to update the GUI. This request is sent to the Event Queue which the GUI thread picks up and makes the appropriate updates. [3]



Limitations of @Future

There are three types of GUI operations. Post-execution (GUI operations that run after a task is complete), Interim (GUI operations that run during the execution of a task), and finalizing (GUI operations that run at the termination of the program). To update the GUI through the @Future annotation, a notifies statement should be defined with the appropriate GUI handler. This allows for the GUI to update immediately after the computation. @Future declarations come with limitations however. @Future only allows for post-execution GUI operations only and does not support GUI handler methods with input arguments. If using @Future, a sequential version of the GUI may not run without code restructuring disallowing a means of validation.

@Gui

@PT allows for all types of GUI operations to be handled asynchronously through the @Gui annotation. Like @Future, @Gui is declared at the invocation of a task, but the task being that of a GUI handler.

A post-execution operation may be defined through a dependency. Like @Future, the dependency can be called explicitly within the annotation through notifiedBy or the return value of an @Future task may be used as an argument for the GUI handler task.

```
@Future
int task1 = foo1();
@Gui(notifiedBy = "task1")<<or>>@Gui
Void updateGui = foo2();          Void updateGui = foo2(task1);
```

Finalizing operations are annotated similarly to post-execution operations.

Interim operations require the implementation of a barrier, disallowing progression of tasks called by @Future until the task itself is finished while continuously running the annotated GUI handler task.

```
@Future
int task1 = foo1();
@Gui
```

```
Void updateGui = foo2(task1);
int barrier = task1;
```

Implementation

To demonstrate the features of @PT, a 'Photomosaic' builder was implemented as a GUI application. A Photomosaic is a recreation of a reference image by replacing each subsection of the image with an image from a selected library that has similar colour values. Photomosaic was chosen because the tasks required in computing a Photomosaic would allow the demonstration of one-off tasks, multi-tasks, IO tasks, GUI updates (interim and post-execution), and async exception handling; all of which are constructs supported by the @PT library.

There are 5 distinct one-off tasks involved in this implementation of Photomosaic creation. These are:

- Downloading images (using the Flickr API) to populate the image library (ImageDownloader)
- Processing the image library and storing them in memory (ImageLibrary)
- Calculating average RGB values for each image in the library (RGBLibrary)
- Partitioning reference image into subsections called cells and then averaging the RGB values for each cell (ImageGrid)
- Substitution of each cell with the image that is closest to its average RGB value by comparing its average RGB value with the entirety of RGBLibrary (MosaicBuilder)

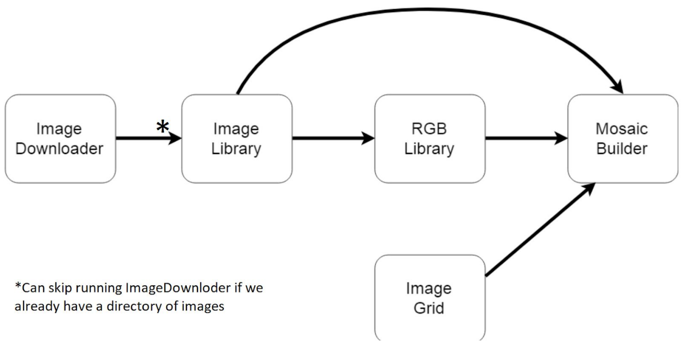
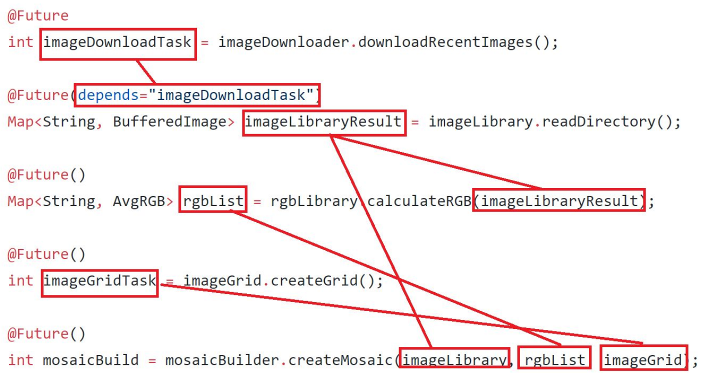


Image Downloader
*
Image Library
RGB Library
Mosaic Builder
Image Grid
*Can skip running ImageDownloder if we already have a directory of images

Parallelising 'one-off tasks'

From these set of tasks, a task graph can be constructed to allow the analysis of dependencies within the tasks. Here is a code snippet of the sequential code made parallel after incorporating @PT annotations:

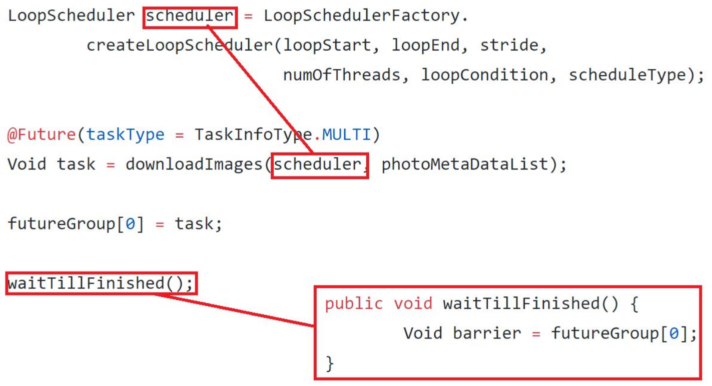


```
@Future
int imageDownloadTask = imageDownloader.downloadRecentImages();

@Future(depends="imageDownloadTask")
Map<String, BufferedImage> imageLibraryResult = imageLibrary.readDirectory();

@Future()
Map<String, AvgRGB> rgbList = rgbLibrary.calculateRGB(imageLibraryResult);

@Future()
int imageGridTask = imageGrid.createGrid();

@Future()
int mosaicBuild = mosaicBuilder.createMosaic(imageLibrary, rgbList, imageGrid);
```

The connecting red boxes indicate a dependency between two tasks. For example, ImageLibrary's dependsOn attribute indicates an explicit dependency on the ImageDownloader task. Another example is RGBLibrary which uses the result of ImageLibrary directly to create an implicit dependency. From these annotations, @PT infers that ImageDownloader and ImageGrid have no dependencies and is therefore able to execute both tasks at the same time. Therefore, a flow of tasks described by the given task graph is shown without restructuring the sequential structure.

Parallelising 'multi-tasks'

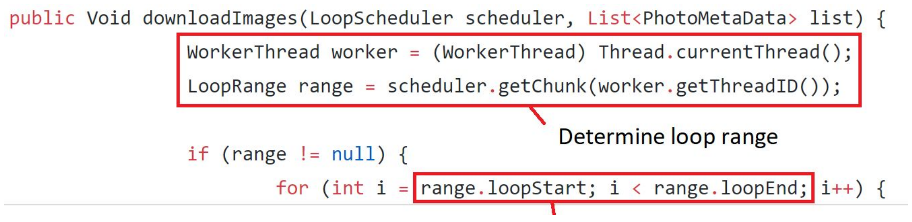
Each of the one-off tasks essentially consists of iterating over and processing a set of data items in a for loop. For example, ImageDownloader iterates over a list of URLs and downloads the image in each one separately. These multi-tasks do not have any inter-loop dependencies which means the loop can be easily split into chunks for each worker thread to process.



```
LoopScheduler scheduler = LoopSchedulerFactory.
        createLoopScheduler(loopStart, loopEnd, stride,
                            numOfThreads, loopCondition, scheduleType);

@Future(taskType = TaskInfoType.MULTI)
Void task = downloadImages(scheduler, photoMetaDataList);

futureGroup[0] = task;

waitTillFinished();
```

```
public void waitTillFinished() {
        Void barrier = futureGroup[0];
}
```

This is a snippet of the ImageDownloader processing code. "downloadImages" is the method containing the for loop. It is annotated with @Future and indicated to be a multi-task. A loop scheduler is also created with information about the loop. At the end, the waitTillFinished() method serves as an implicit barrier by directly using the return value of the multi-task. The scheduler is passed into the multi-task method invocation as to allow worker threads to determine their allocated loop chunk, as seen in the code snippet below:

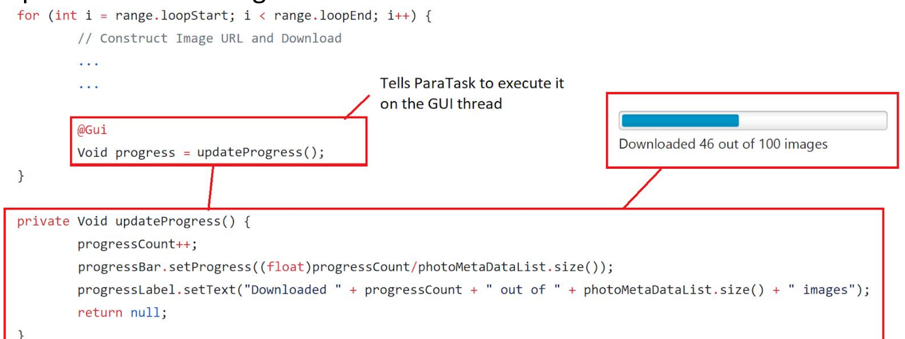


```
public Void downloadImages(LoopScheduler scheduler, List<PhotoMetaData> list) {
        WorkerThread worker = (WorkerThread) Thread.currentThread();
        LoopRange range = scheduler.getChunk(worker.getThreadID());

        if (range != null) {
                for (int i = range.loopStart; i < range.loopEnd; i++) {
```

Parallelising GUI updates

The GUI for the application was developed using JavaFX. The ParaTask runtime currently does not support the scheduling of tasks on JavaFX's application thread (equivalent to Swing's EDT). In order to circumvent this, the source code for the ParaTask runtime is modified such that it schedules tasks on JavaFX's application thread rather than Swing's EDT.

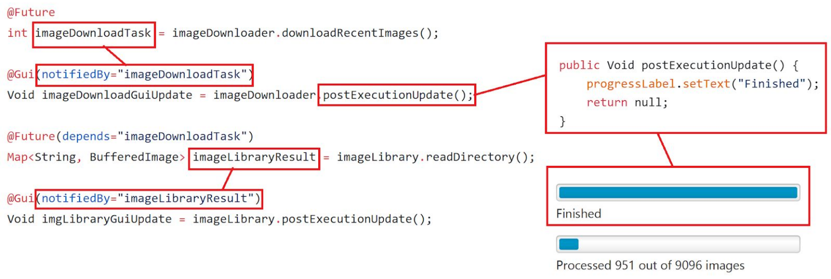
The application incorporates two forms of GUI updates: interim and post-execution updates. Interim updates are updates to the GUI to indicate the current progress of a particular task. Interim updates are implemented in the application by updating a progress bar and label after every iteration of the multi-task loop as shown in the figure below:



```
for (int i = range.loopStart; i < range.loopEnd; i++) {
        // Construct Image URL and Download
        ...
        ...

        @Gui
        Void progress = updateProgress();
}

private Void updateProgress() {
        progressCount++;
        progressBar.setProgress((float)progressCount/photoMetaDataList.size());
        progressLabel.setText("Downloaded " + progressCount + " out of " + photoMetaDataList.size() + " images");
        return null;
}
```

Tells ParaTask to execute it on the GUI thread
Downloaded 46 out of 100 images

Post-execution updates are demonstrated by changing a task's progress label to say 'Finish' when it has finished executing. This is shown in the figure below:



```
@Future
int imageDownloadTask = imageDownloader.downloadRecentImages();

@Gui(notifiedBy="imageDownloadTask")
Void imageDownloadGuiUpdate = imageDownloader.postExecutionUpdate();

@Future(depends="imageDownloadTask")
Map<String, BufferedImage> imageLibraryResult = imageLibrary.readDirectory();

@Gui(notifiedBy="imageLibraryResult")
Void imgLibraryGuiUpdate = imageLibrary.postExecutionUpdate();
```

```
public Void postExecutionUpdate() {
    progressLabel.setText("Finished");
    return null;
}
```

A note has to be made of the 'notifiedBy' attribute in the @Gui annotation. This indicates that the update should not be queued on the GUI thread until the indicated task has finished executing. Through the use of @Gui, interim and post-execution GUI updates have been implemented without requiring significant restructuring of the sequential code whilst ensuring that the updates are scheduled on the GUI thread.

Parallelising 'IO Tasks'

The code snippet below demonstrates the parallelisation of IO tasks. The code is executed when the user clicks on the 'save to disk' button. This will save the Photomosaic to disk, which takes a considerable amount of time due to the size of the image. The post-execution update re-enables the 'save' button once the image has been saved to disk.

```
if (saveToFile != null) {
        ((Button)arg0.getSource()).setDisable(true);

        @Future(taskType = TaskInfoType.INTERACTIVE)
        int ioTask = mosaicBuilder.saveImageOnDisk(saveToFile);

        @Gui(notifiedBy = "ioTask")
        Void enableSave = mosaicBuilder.enableSave((Button)arg0.getSource());
}
```

Asynchronous Exception Handling

A problem with creating the Photomosaic is that the resulting image could be too large to fit onto memory, thereby throwing an OutOfMemoryError. To prevent this, the number of pixels the final result will have is calculated and a custom exception is thrown if it exceeds a predetermined amount. In order to catch this exception, the method is annotated with an @AsyncCatch annotation, indicating the thrown exception and a method to call once it has been caught. This handler indicates to the user that their Mosaic is too large to build, prompting them to adjust their input as to reduce the size to an appropriate level.

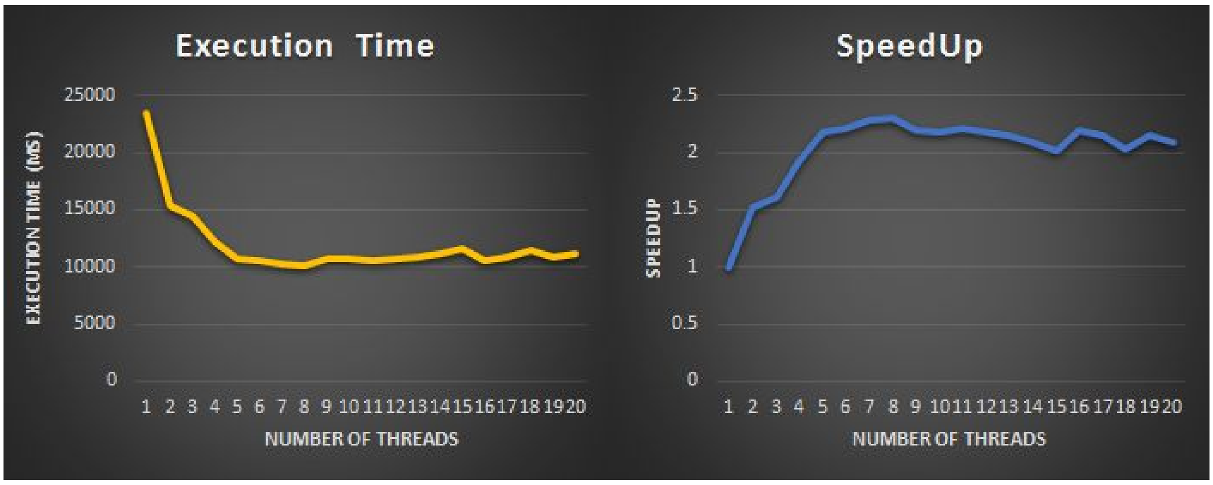
```
@AsyncCatch(throwables= {ImageTooBigException.class}, handlers= {"handleImageTooBig()"})
@Future()
int mosaicBuild = mosaicBuilder.createMosaic(imageLibrary, rgbList, imageGrid);
```

Challenges

A particular challenge in using @PT was the initial setup of it. Due to a lack of documentation, it was difficult to understand how @PT worked and how to set it up. Initially, it was assumed that the annotations would be processed in runtime. However, after wondering why the code was not parallelised, the build.xml file was investigated closely and realised that the annotated code must be parsed through Spoon to generate the parallel code. In order to make development and testing easier, an additional Ant build is created to automatically compile and run the generated code.

Evaluation

Speed up



When running the computations for producing a mosaic in parallel execution, it produces the execution times shown in the above graph for each number of threads. Using these execution times, the speed up for each number of threads can be observed. From 1 thread onwards, speed up significantly increases until it peaks at about 7-8 threads where speed up starts to slow down slightly. Even though the computer this application was running on only had 4 processors, an increase in speed up is still observed with more than 4 threads. This may be due to following reasons:

1) Calling for an image download requiring wait times for the request to reach the server and wait times for the image data to arrive from the server: Waiting does not require CPU, so the thread ends up with nothing to do. Meanwhile, the processor switches another runnable thread and calls for another image download. This repeats for as long as a thread still has images to download. This means there are more images downloading at once because the processors spend less time in the context of a waiting thread.

2) Relation to hyperthreading of the Intel processors on the machine.

A processor can achieve hyperthreading by concurrently executing 2 threads, allowing the processors to do more work within each clock cycle. This results in the operating system viewing the hyperthreading processor as two separate processors, thus is able to allocate two threads to it.

Conclusion

The use of @PT has made parallelising sequential code much easier. This is mainly due to the fact that there is little to no restructuring of the sequential code required. It also provides a safe and easy way to perform GUI updates and handle exceptions through additional annotations. Despite the initial set up difficulties, @PT has proven to be a great library for developing parallel code that is easily readable whilst providing a huge speed up benefit with minimal effort.