Project 9: Creating a Simple GUI Application using @PT

Group 14: Andy Tang, Joseph Dumogho, Lite Kim

Outline

- Introduction
- @PT
- Implementation: Photomosaic
- Parallelisation of Application using @PT
- Demo
- Future Work

Develop a GUI Application

- Written in Java
- Uses @PT (Annotated Parallel Task)
- Includes API
- IO Tasks
- Multi-tasks
- One-off Computations
- Sequential and Parallel modes

Annotated Parallel Task (@PT)

Standard Parallelization

Extended Libraries

- Pros
  - Portability
- Cons
  - Heavy restructuring of sequential code
  - Potential inconsistent implementation
  - Difficult to understand
  - Difficulties in modification

Language Constructs

- Pros
  - Hide complexity of parallelization
  - Avoid boilerplate code
- Cons
  - Prior concept understanding
  - Different compilation requirements
  - Loses portability

Annotated Parallel Task (@PT)

Motivation

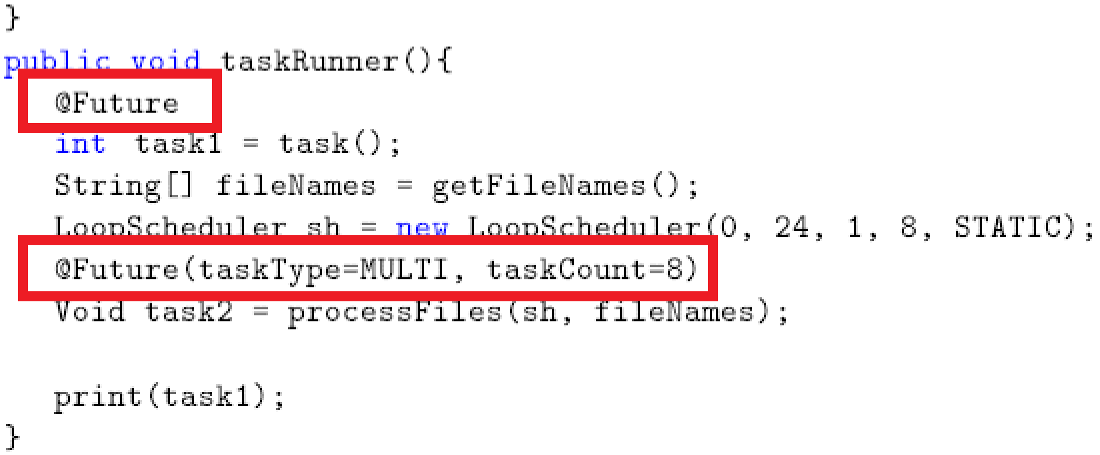
- Retain the pros and build upon the cons of existing parallelization concepts
    - Benefit development during programming
    - Minimize original sequential structure
    - Portable and universal

@PT

- Language construct framework
  - Java annotations
  - Hides complexities of parallelization
  - Avoid boilerplate
- Object Oriented approach on asynchronous execution
- Focus on GUI-responsiveness

@PT Annotations

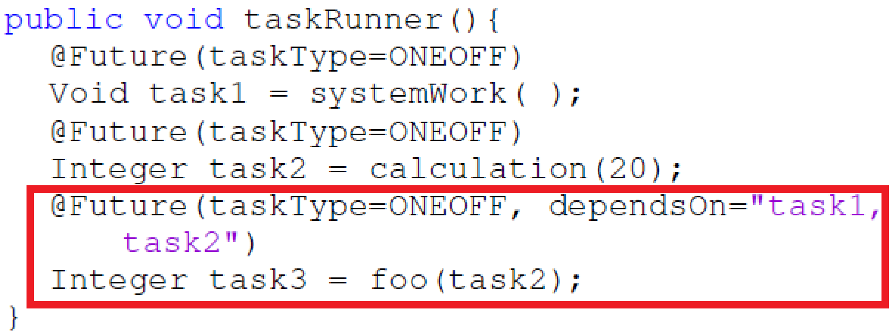
- @Future
    - ○ taskType
    - ○ taskCount



```
}
public void taskRunner(){
  @Future
  int  task1 = task();
  String[] fileNames = getFileNames();
  LoopScheduler sh = new LoopScheduler(0, 24, 1, 8, STATIC);
  @Future(taskType=MULTI, taskCount=8)
  Void task2 = processFiles(sh, fileNames);

  print(task1);
}
```

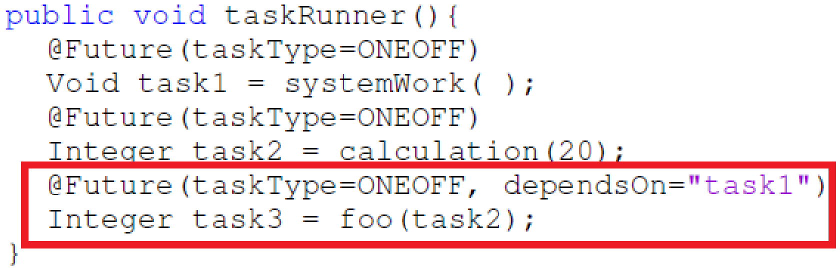
@PT Annotations

- @Future
    - taskType
    - taskCount
    - **dependsOn**



```
public void taskRunner(){
   @Future(taskType=ONEOFF)
   Void task1 = systemWork( );
   @Future(taskType=ONEOFF)
   Integer task2 = calculation(20);
   @Future(taskType=ONEOFF, dependsOn="task1,
       task2")
   Integer task3 = foo(task2);
}
```

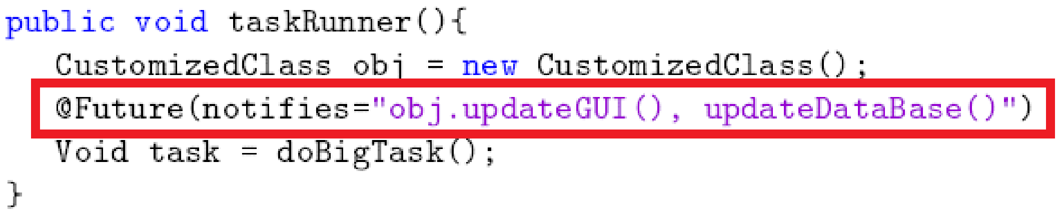
=



```
public void taskRunner(){
   @Future(taskType=ONEOFF)
   Void task1 = systemWork( );
   @Future(taskType=ONEOFF)
   Integer task2 = calculation(20);
   @Future(taskType=ONEOFF, dependsOn="task1")
   Integer task3 = foo(task2);
}
```

@PT Annotations

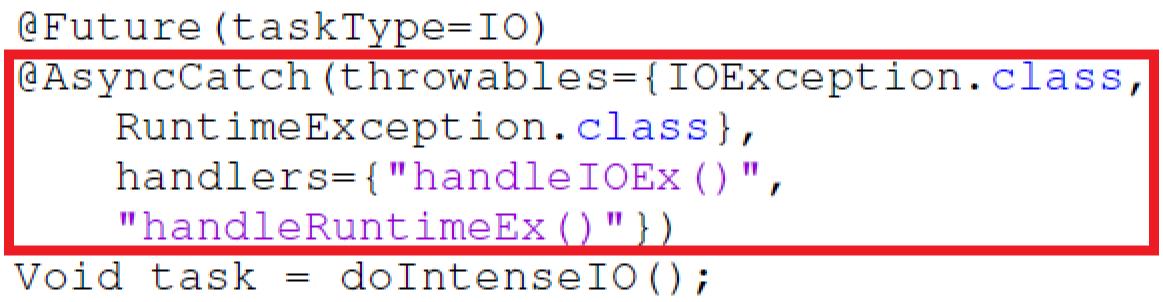
- @Future
  - taskType
  - taskCount
  - dependsOn
  - **notifies**



```
public void taskRunner(){
    CustomizedClass obj = new CustomizedClass();
    @Future(notifies="obj.updateGUI(), updateDataBase()")
    Void task = doBigTask();
}
```

@PT Annotations

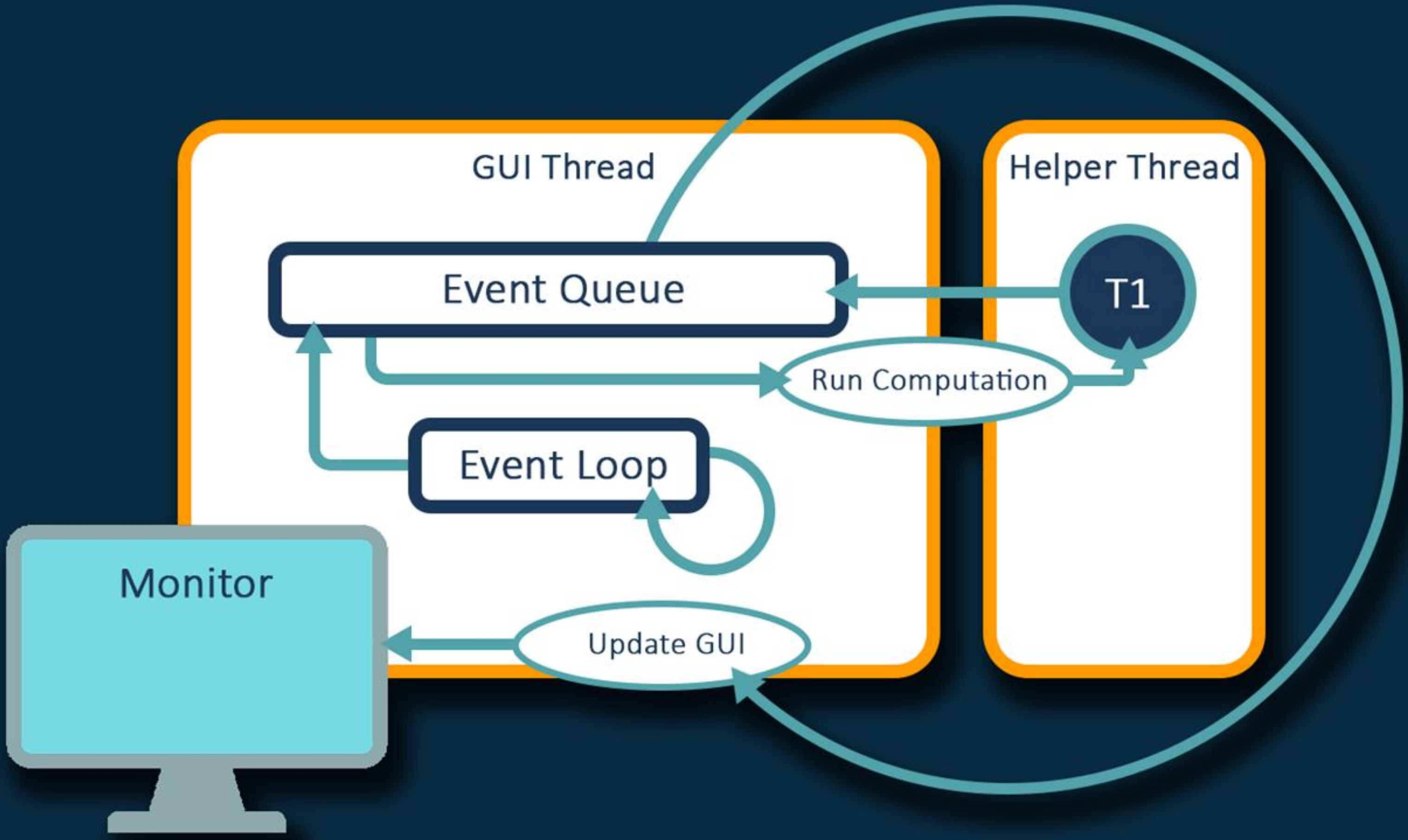
- @Future
  - taskType
  - taskCount
  - dependsOn
  - notifies
- @AsyncCatch
  - **throwables**
    - **Exception classes**
  - **handlers**
    - **Methods for handling**



```
@Future(taskType=IO)
@AsyncCatch(throwables={IOException.class,
    RuntimeException.class},
    handlers={"handleIOEx()",
    "handleRuntimeEx()"})
Void task = doIntenseIO();
```

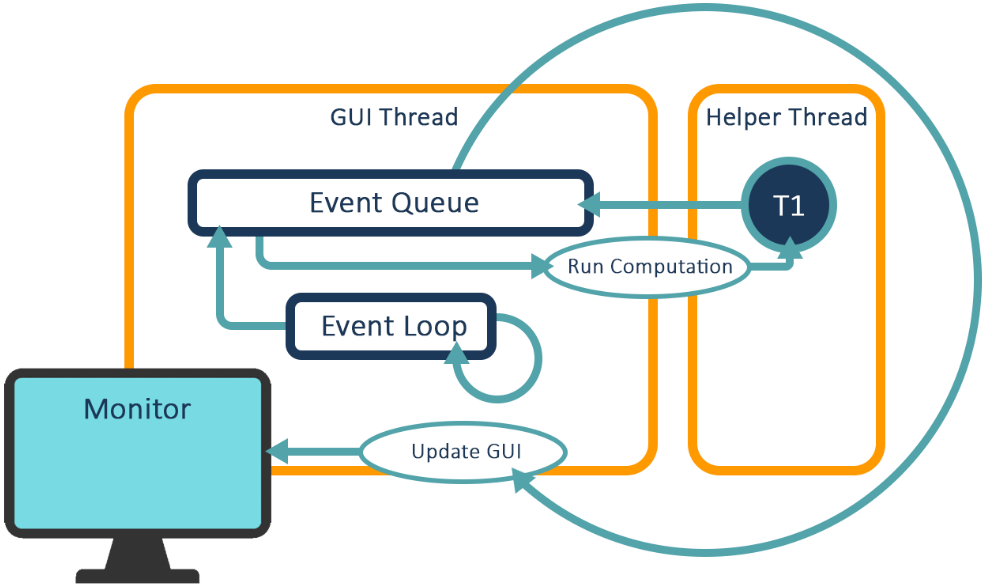
GUI Parallelization

- Provide better user experience (an always interactive GUI)
    - Allocate separate threads
        - GUI Thread (Continuously running)
        - Helper/Computation Thread (Run in background)
    - Thread safety
        - Helper threads should not update GUI
        - One thread updates GUI


GUI Thread
Helper Thread
Event Queue
T1
Run Computation
Event Loop
Monitor
Update GUI

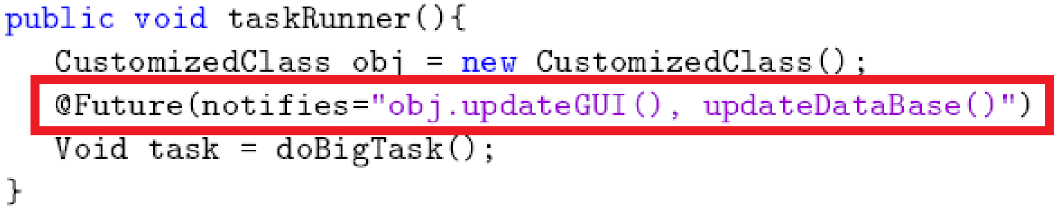
GUI Parallelization

- Considerations
  - 3 types of operations
    - Post-execution
    - Interim
    - Finalizing
  - Timing of GUI update
    - Immediate response



Limitations of @Future

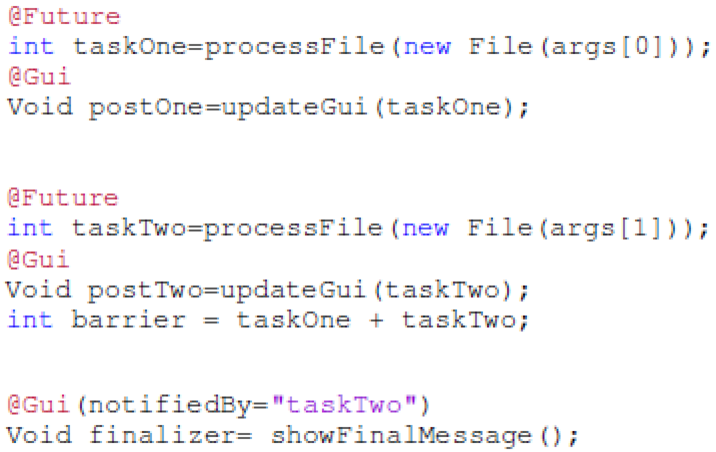
- Allows for post-execution GUI operations only
- GUI methods could not take arguments
- Can not run in sequential (with annotations)
    - No validation



```
public void taskRunner(){
   CustomizedClass obj = new CustomizedClass();
   @Future(notifies="obj.updateGUI(), updateDataBase()")
   Void task = doBigTask();
}
```

@Gui

- @Gui
  - Declared before the GUI method
- Interim Operations
  - Use of a barrier
- Explicitly call dependency in annotation
  - notifiedBy

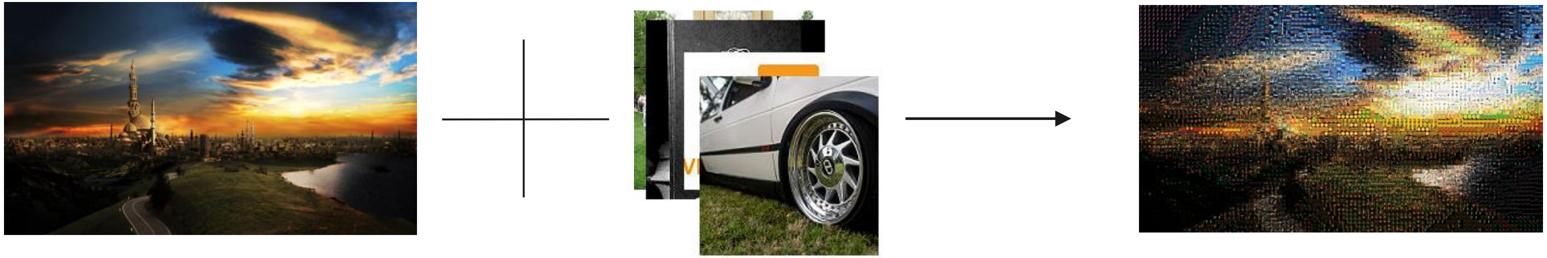


```
@Future
int taskOne=processFile(new File(args[0]));
@Gui
Void postOne=updateGui(taskOne);


@Future
int taskTwo=processFile(new File(args[1]));
@Gui
Void postTwo=updateGui(taskTwo);
int barrier = taskOne + taskTwo;


@Gui(notifiedBy="taskTwo")
Void finalizer= showFinalMessage();
```

Our Implementation: Photomosaic

Photomosaic

- Recreation of reference image by replacing subsections of the image with another image with similar colour values
- Why Photomosaic?
    - Allows us to demonstrate the parallelisation of one-off, multi, and IO tasks
    - Looks cool

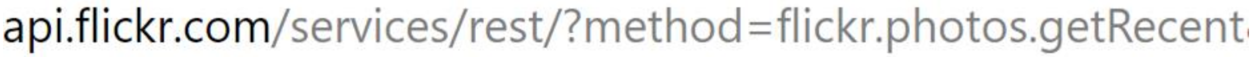


Processing Tasks

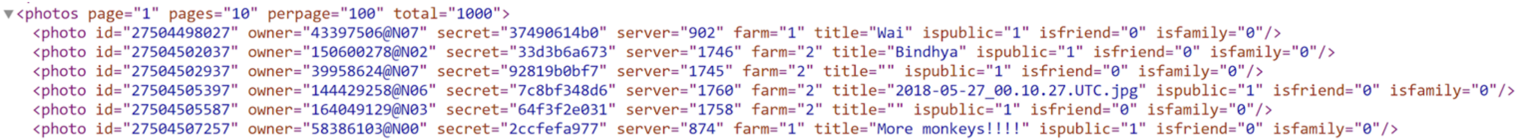
1. Downloading images using Flickr API (ImageDownloader)
2. Processing and storing downloaded images in memory (ImageLibrary)
3. Calculating average RGB values for downloaded images (RGBLibrary)
4. Partitioning reference image into cells to create an ‘Image Grid’ where each cell is the average RGB value of the block of pixels it represents. (ImageGrid)
5. Substitution of each cell with the image that is closest to its average RGB value (MosaicBuilder)

ImageDownloader

- Uses Flickr API to retrieve information about the 100 most recent publicly uploaded photos



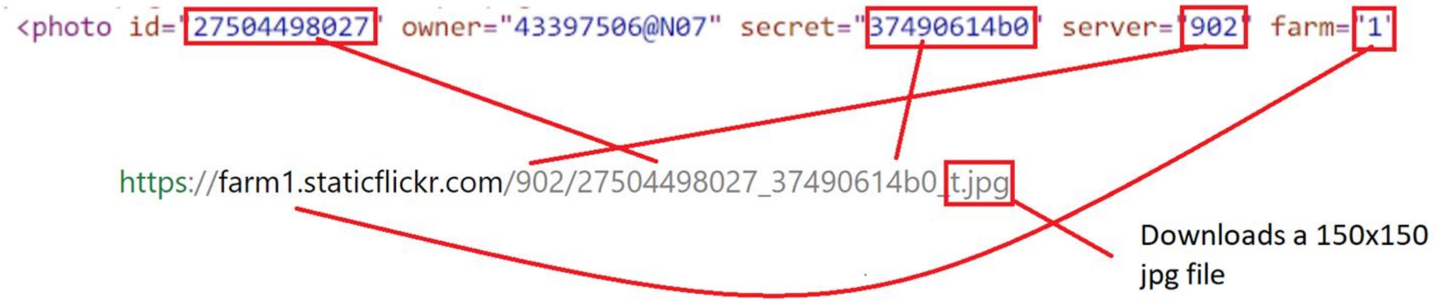
api.flickr.com/services/rest/?method=flickr.photos.getRecent



```
▼<photos page="1" pages="10" perpage="100" total="1000">
   <photo id="27504498027" owner="43397506@N07" secret="37490614b0" server="902" farm="1" title="Wai" ispublic="1" isfriend="0" isfamily="0"/>
   <photo id="27504502037" owner="150600278@N02" secret="33d3b6a673" server="1746" farm="2" title="Bindhya" ispublic="1" isfriend="0" isfamily="0"/>
   <photo id="27504502937" owner="39958624@N07" secret="92819b0bf7" server="1745" farm="2" title="" ispublic="1" isfriend="0" isfamily="0"/>
   <photo id="27504505397" owner="144429258@N06" secret="7c8bf348d6" server="1760" farm="2" title="2018-05-27_00.10.27.UTC.jpg" ispublic="1" isfriend="0" isfamily="0"/>
   <photo id="27504505587" owner="164049129@N03" secret="64f3f2e031" server="1758" farm="2" title="" ispublic="1" isfriend="0" isfamily="0"/>
   <photo id="27504507257" owner="58386103@N00" secret="2ccfefa977" server="874" farm="1" title="More monkeys!!!!" ispublic="1" isfriend="0" isfamily="0"/>
```

ImageDownloader

- Uses the return XML data to construct a URL to the direct image which we could then download and save to disk



```
<photo id='27504498027' owner="43397506@N07" secret="37490614b0' server='902' farm='1'
```

https://farm1.staticflickr.com/902/27504498027_37490614b0_t.jpg

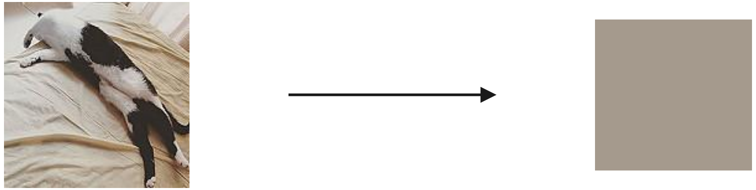
Downloads a 150x150 jpg file

ImageLibrary

- Processes the downloaded images and stores them in a HashMap<String, BufferedImage>
- Key: String of the image’s file name
- Value: BufferedImage of the downloaded image
- Allows downscaling of the downloaded images
- Removes any images that are not the right size (150x150 or downscaled resolution)

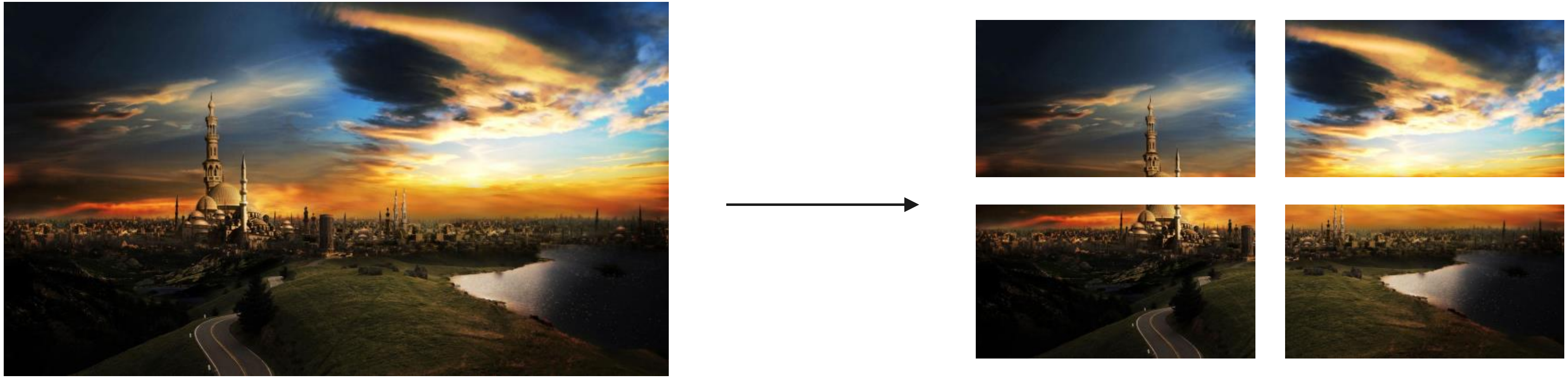
RGBLibrary

- Calculates the average RGB value for each image and stores it in a HashMap<String, AvgRGB>
- Key: String of the image's file name
- Value: AvgRGB object that stores the averaged RGB value
- Procedure:
    - For each image in ImageLibrary:
    - Sum up the independent R, G, and B values of every pixel in the image
    - Divide the sums by the number of pixels to get the average for R, G, and B.
    - Store the values as an AvgRGB object in the HashMap



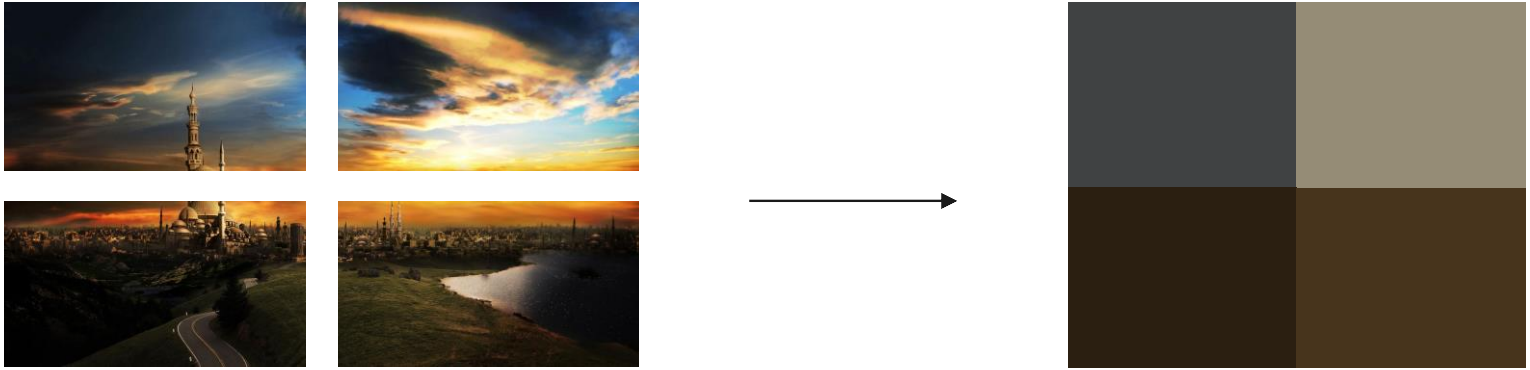
ImageGrid

- Partition reference image into cells
- Each cell represents a subsection of the image



ImageGrid

- Calculate the average RGB value for each cell
- Store results in a 2D array of AvgRGB objects
- 2D array is essentially another ‘image’ where each pixel represents the average RGB value for that cell in the reference image



MosaicBuilder

- First create blank BufferedImage with dimensions:

$$width = \underbrace{\boxed{\frac{refImageWidth}{cellWidth}}}_{\text{ImageGrid width}} * downloadedImageWidth$$

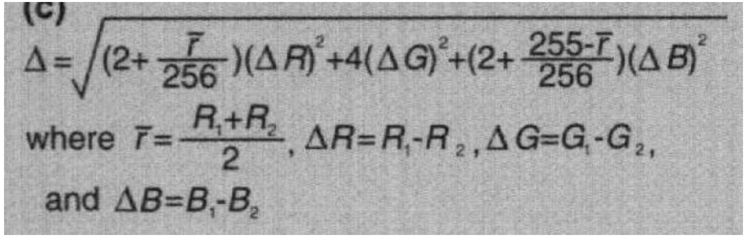
$$height = \underbrace{\boxed{\frac{refImageHeight}{cellHeight}}}_{\text{ImageGrid height}} * downloadedImageHeight$$

MosaicBuilder

- Compare each ImageGrid cell with every AvgRGB object in RGBLibrary to find the closest matching one
- Euclidean Distance Algorithm:

$$dist = \sqrt{(r_1 - r_2)^2 + (g_1 - g_2)^2 + (b_1 - b_2)^2}$$

- Riemersma metric:



(c)

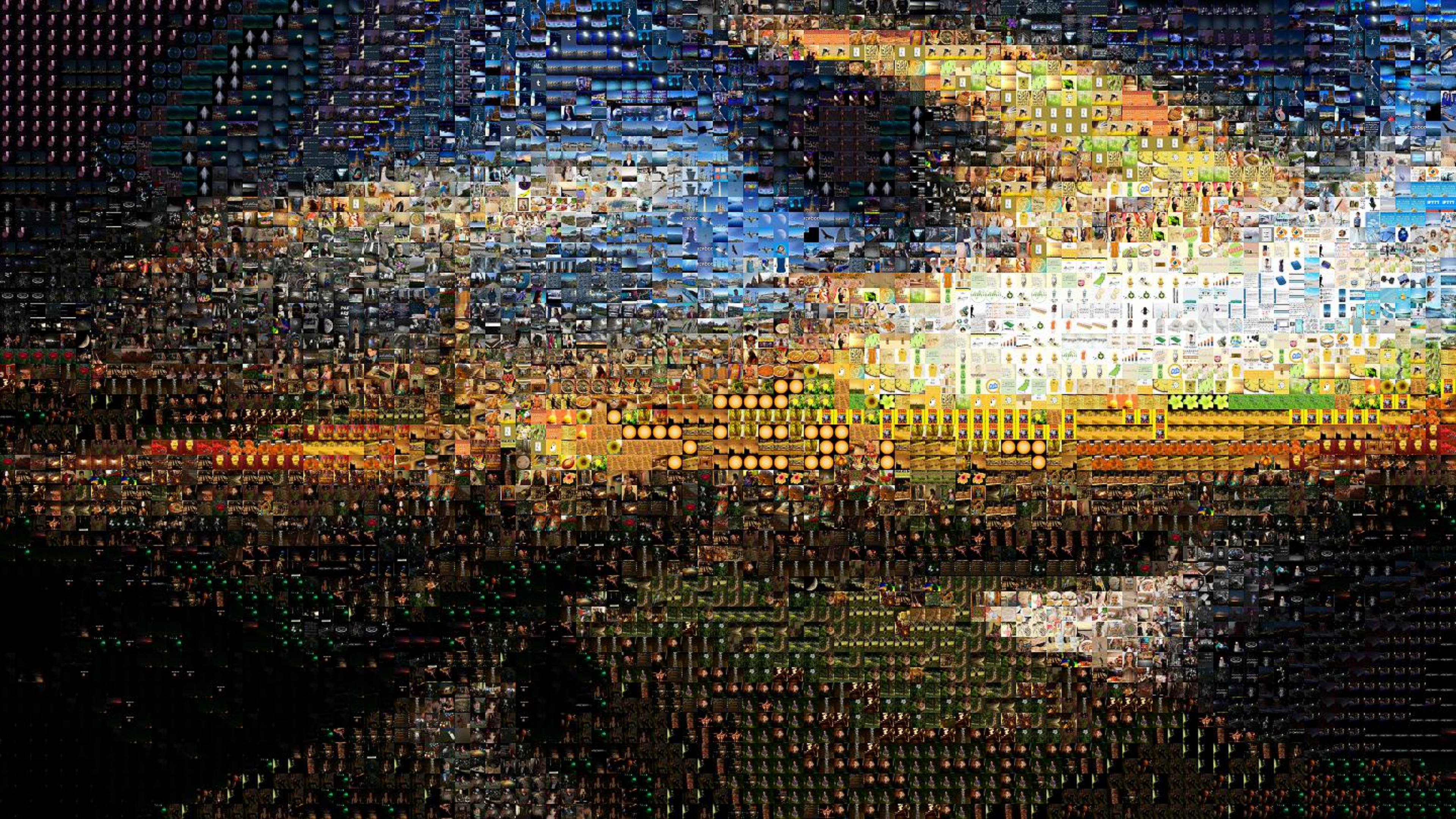
$$\Delta = \sqrt{(2+\frac{\bar{r}}{256})(\Delta R)^2 + 4(\Delta G)^2 + (2+\frac{255-\bar{r}}{256})(\Delta B)^2}$$

where $\bar{r} = \frac{R_1+R_2}{2}$, $\Delta R = R_1 - R_2$, $\Delta G = G_1 - G_2$,

and $\Delta B = B_1 - B_2$

MosaicBuilder

- When the best match has been found, retrieve the corresponding BufferedImage in ImageLibrary using it’s file name
- Draw the matching BufferedImage into the new BufferedImage at its appropriate position
- Repeat until all cells have been substituted for an image in ImageLibrary



Improving the output

- How well the resulting Photomosaic resembles the original reference image changes depending on a few things:
    - The amount of images in ImageLibrary
    - The size of each cell in ImageGrid
- Making it look ‘better’ means increasing its run time and memory usage

Parallelising the Application using @PT

Parallelisation Levels

- Three levels of parallelisation in our application
    - Parallelising each individual task (Data parallelism)
    - Parallelising the tasks as a whole (Task parallelism)
    - Parallelising the GUI (Intermittent updates and Post-execution GUI updates)

Parallelising individual tasks

- Each task consists of iterating over and processing a set of data items (data parallelism)
- E.g. ImageDownloader is just iterating over a list of URLs and downloading the image from each one
- This means we can easily parallelise each individual task as there are no inter-loop dependences
- These are called multi-tasks in @PT

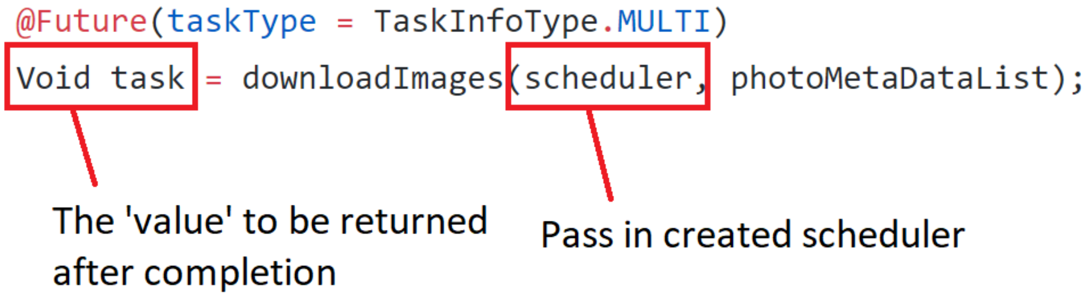
Parallelising individual tasks

- First create loop scheduler

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

Parallelising individual tasks

- Annotate multi-task with @Future



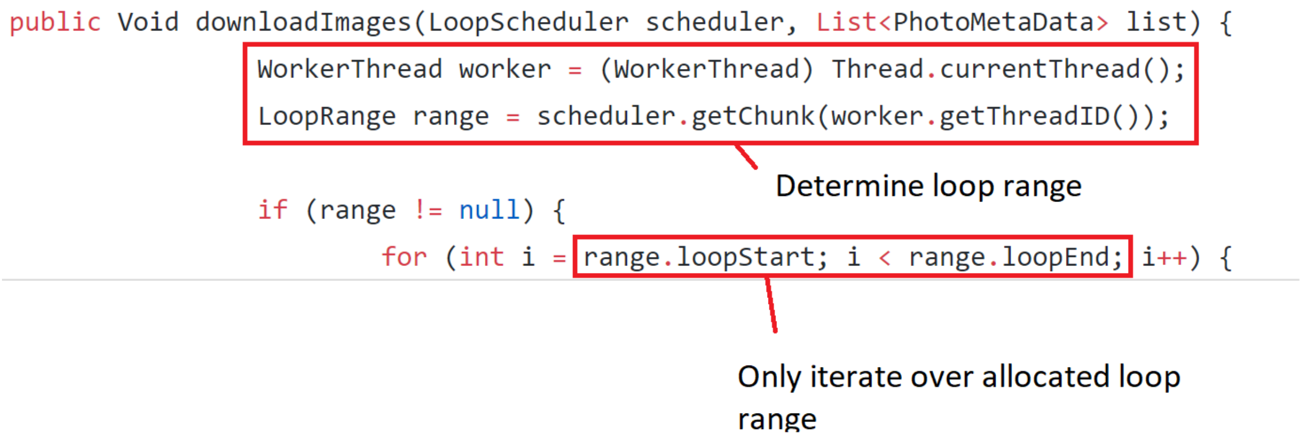
```
@Future(taskType = TaskInfoType.MULTI)
Void task = downloadImages(scheduler, photoMetaDataList);
```

The 'value' to be returned after completion

Pass in created scheduler

Parallelising individual tasks

- Modify loop to iterate over allocated chunk



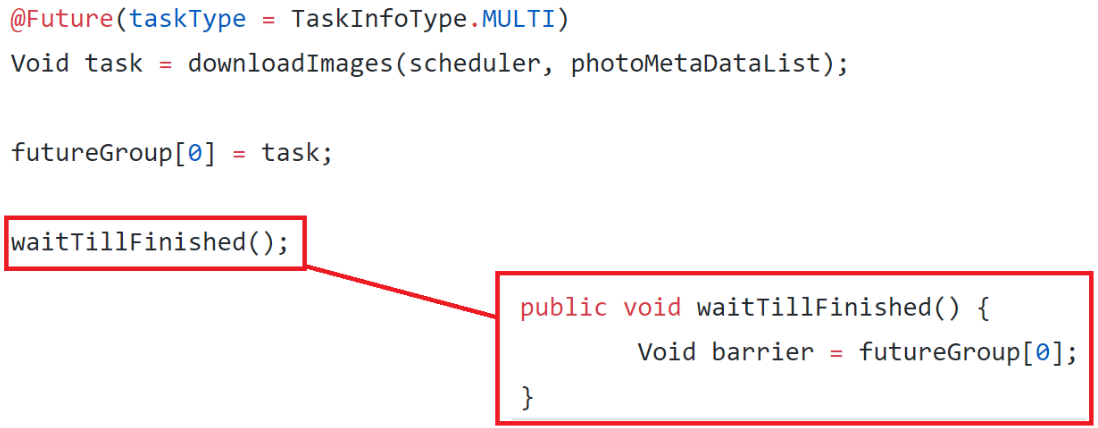
```
public Void downloadImages(LoopScheduler scheduler, List<PhotoMetaData> list) {
            WorkerThread worker = (WorkerThread) Thread.currentThread();
            LoopRange range = scheduler.getChunk(worker.getThreadID());

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

Determine loop range

Only iterate over allocated loop range

Parallelising individual tasks

- Create implicit barrier

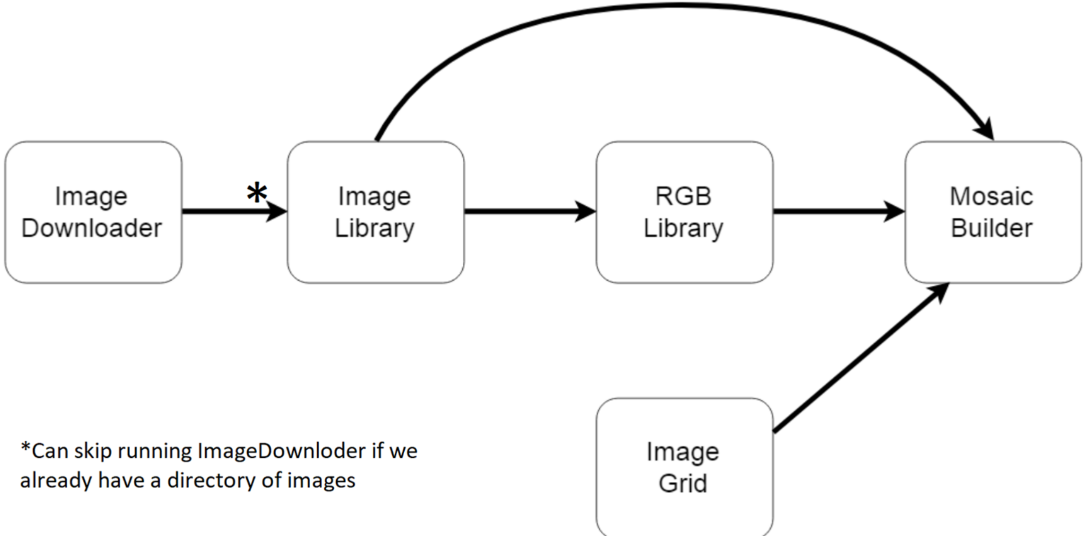


```
@Future(taskType = TaskInfoType.MULTI)
Void task = downloadImages(scheduler, photoMetaDataList);

futureGroup[0] = task;

waitTillFinished();
```

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

Parallelising the tasks as a whole

- Construct task graph so we're aware of the dependences



*Can skip running ImageDownloder if we already have a directory of images

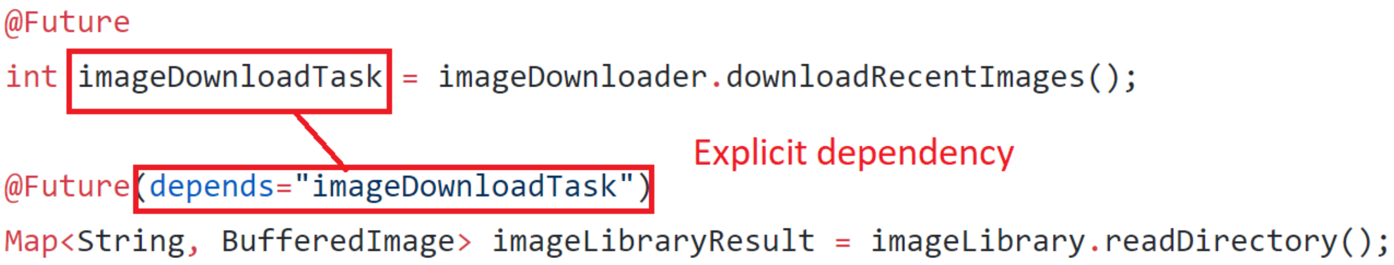
Parallelising the tasks as a whole

- Now just annotate the code accordingly with @Future

```
@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);
```

Explicit Dependencies

- Define explicit dependencies by using the return variable’s name in the @Future annotation of the task with the dependency

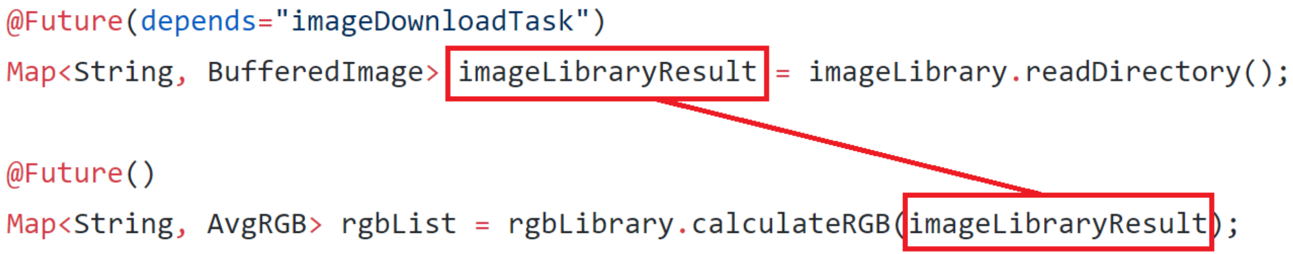


```
@Future
int imageDownloadTask = imageDownloader.downloadRecentImages();

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

Explicit dependency

Implicit Dependencies

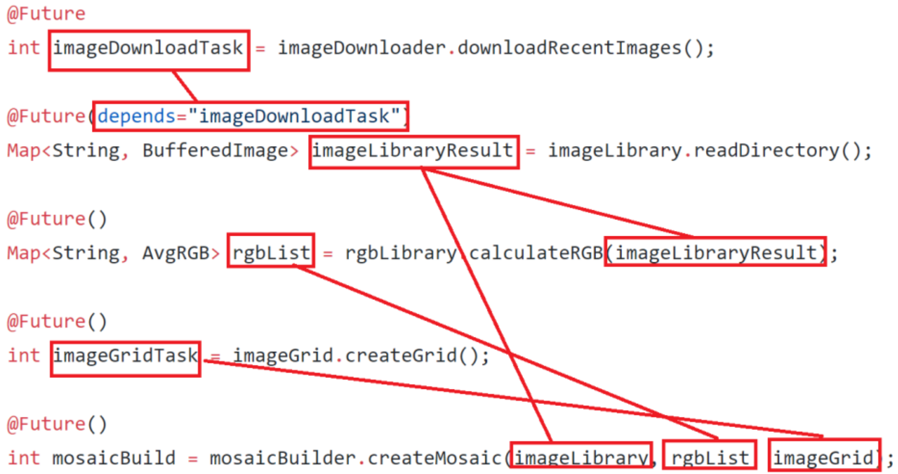
- If a Task relies on the result of another task, you can pass in the return variable directly to indicate that it is an implicit dependency



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

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

Implicit Parallelisation of Tasks

- ImageGrid and ImageDownloader has no implicit or explicit dependencies, so ParaTask will execute both these tasks at the same time.



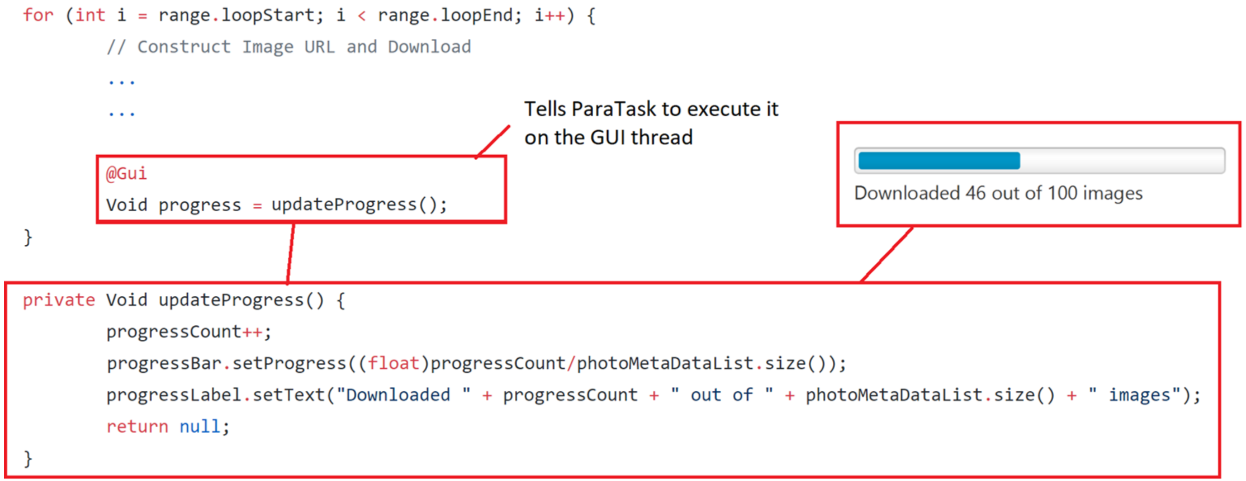
```
@Future
int imageDownloadTask = imageDownloader.downloadRecentImages();

@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);
```

Parallelising the GUI

- Two kinds of GUI updates
  - Intermittent Updates
  - Post-execution Updates

Intermittent Updates

- Updates to the GUI to indicate the current progress of a task



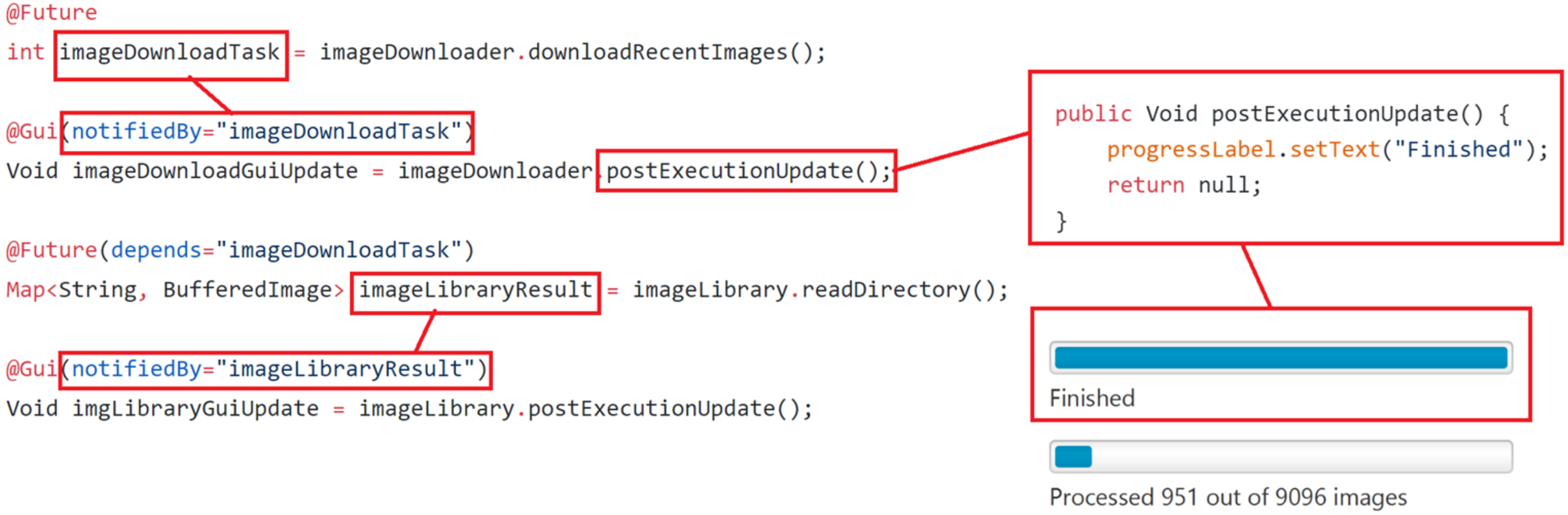
```
for (int i = range.loopStart; i < range.loopEnd; i++) {
        // Construct Image URL and Download
        ...
        ...

        @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

- Updates the GUI when a task has finished executing



```
@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;
}
```

Finished

Processed 951 out of 9096 images

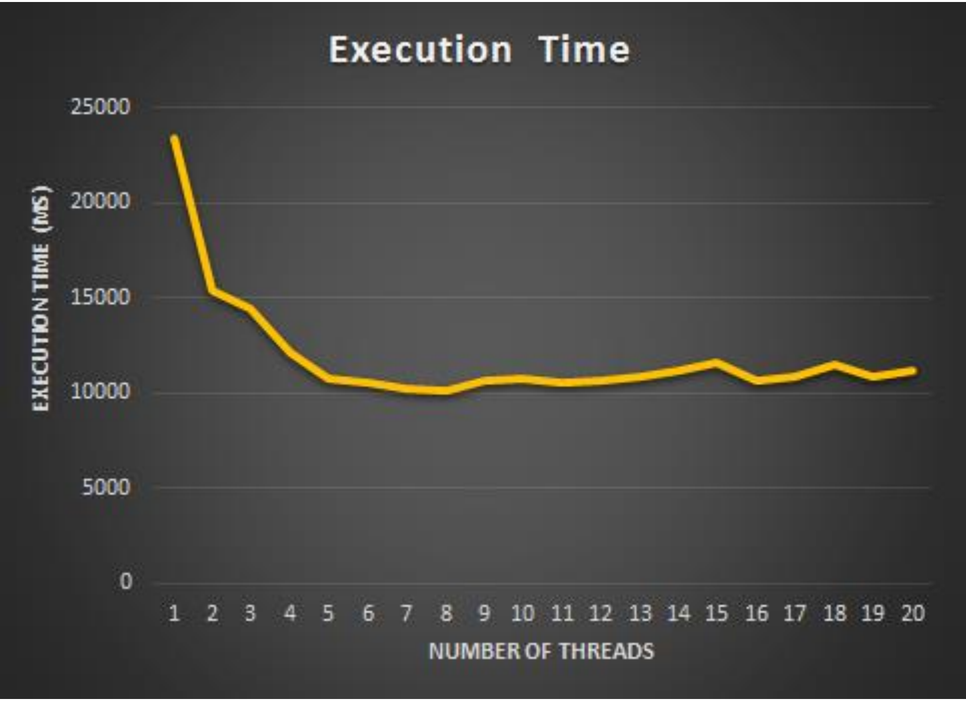
Demo Time!

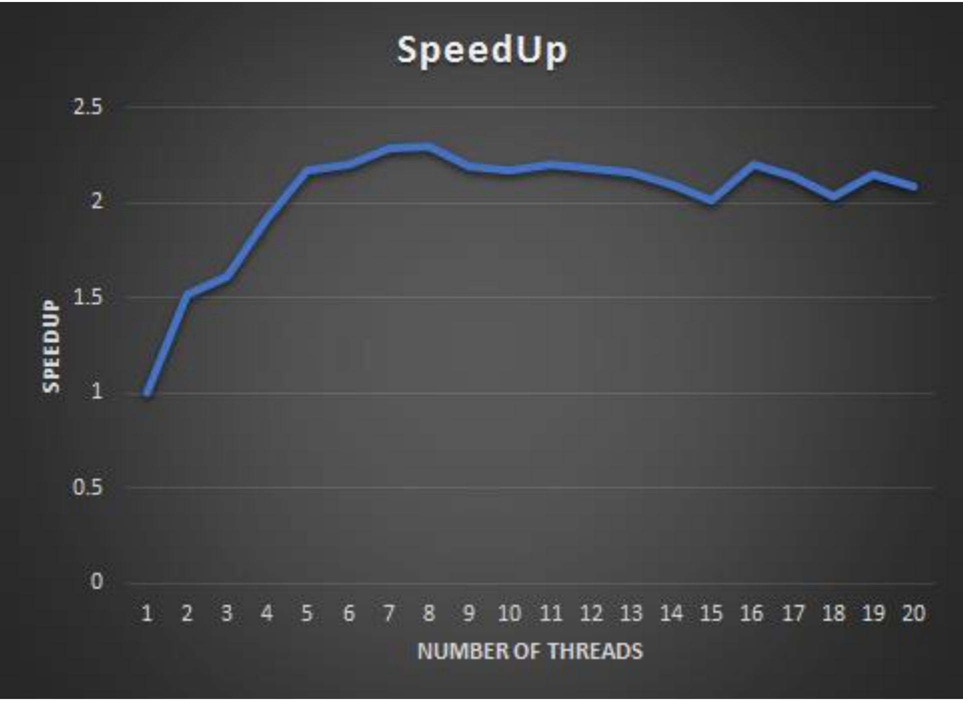
GUI Walkthrough

Demonstration

- Sequential
- Parallel
- No dedicated Gui thread

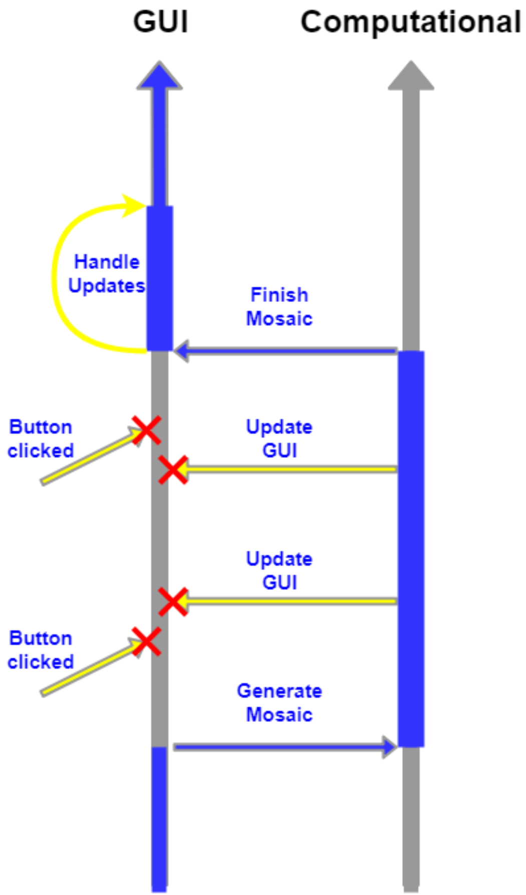
Speedup


Execution Time


SpeedUp

Problems with no dedicated Gui thread

- Gui becomes unresponsive
  - The only existing thread is computing and generating the mosaic
  - No thread is managing the Gui
- Updates don’t appear
  - The computing thread is sending updates to the Gui but there are no threads to handle the updates.
  - Updates only handled after generating the photomosaic.
  - User has no way of knowing the progress of the mosaic generation



Future Work

Fixing Application problems

- Gui can’t update fast enough when building a large mosaic.
  - If a solution isn’t possible:
    - Predict the size of computation
    - Don’t show canvas until the mosaic is finished if the mosaic is too large for Gui

Applying Other High Performance Computing concepts

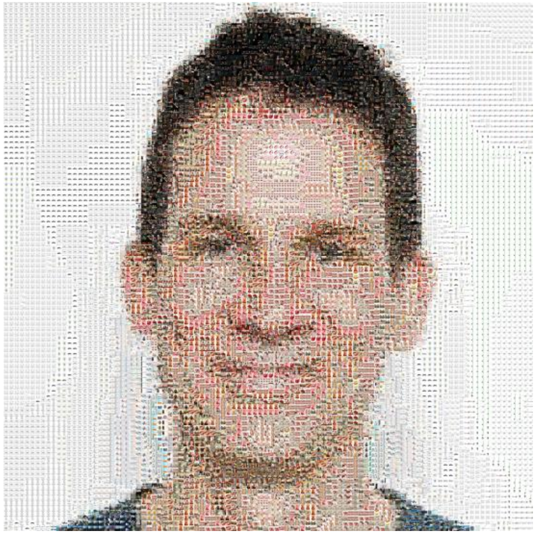
- Benchmarking Code
  - evaluate its performance
  - identify computational bottlenecks
- Work Stealing
  - Task allocation is more evenly distributed
- GPU Image Processing
  - GPU is highly optimized for image processing
- Cloud Computing
  - Image processing delegated to server
  - Likely to be faster

Other Photomosaic implementations

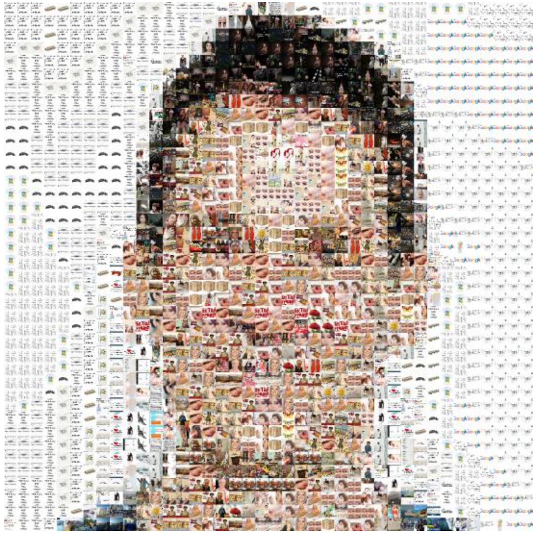
- Simple photomosaic - our implementation
- Advanced Photomosaic - better results
- False Photomosaic - fake!

Simple Photomosaic

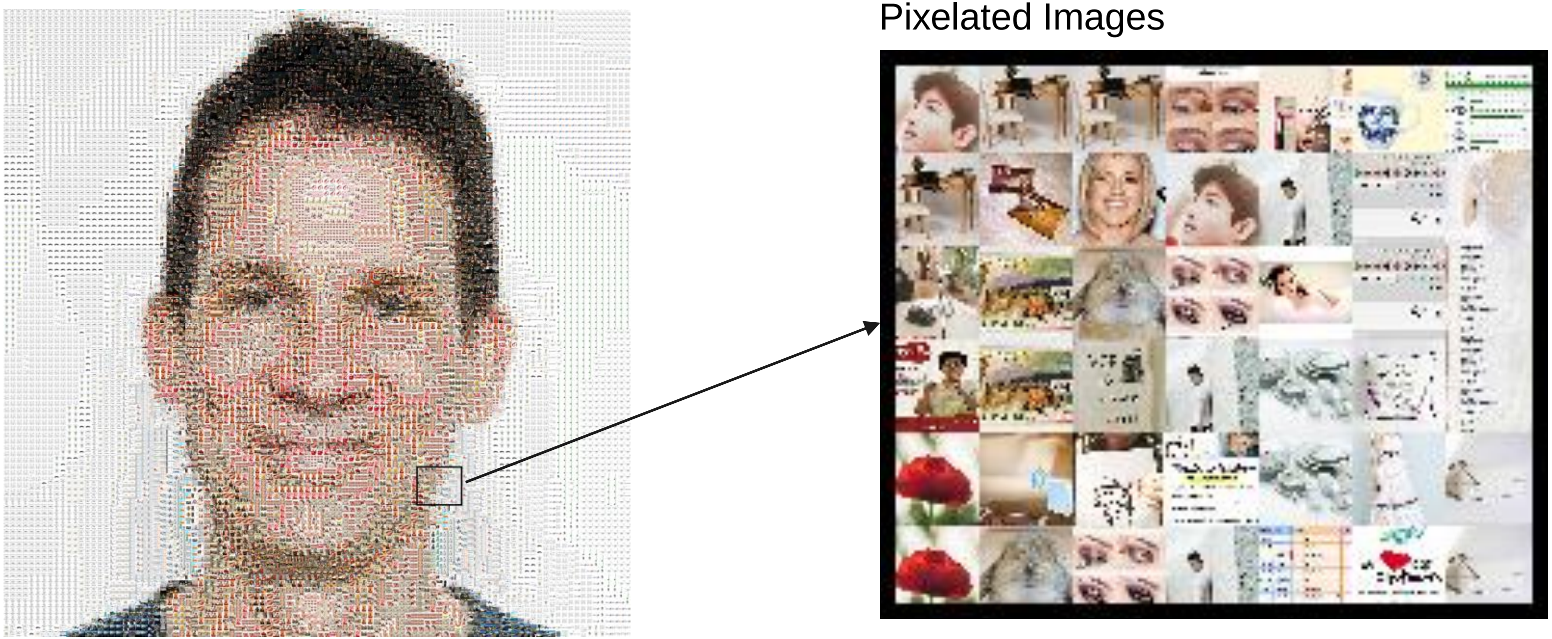
Coherent but small cell images



Large cell images but incoherent



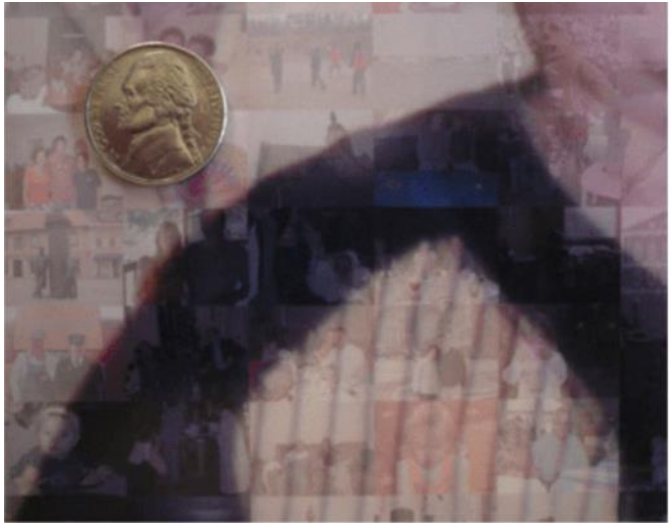
Simple Photomosaic



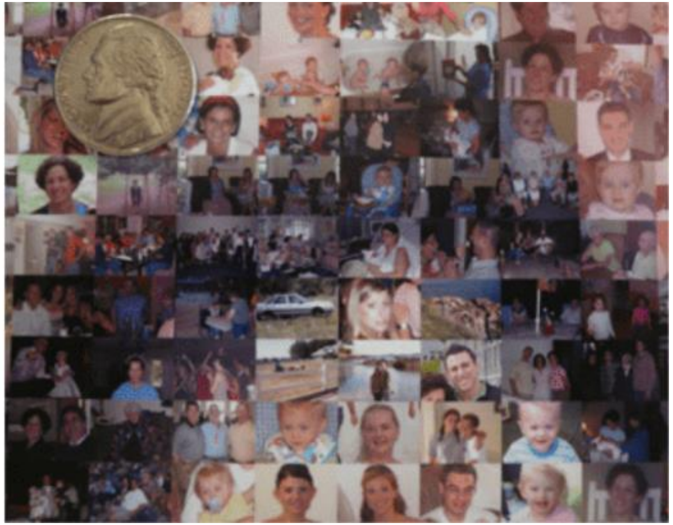
Pixelated Images

False Photomosaics

False Photomosaic

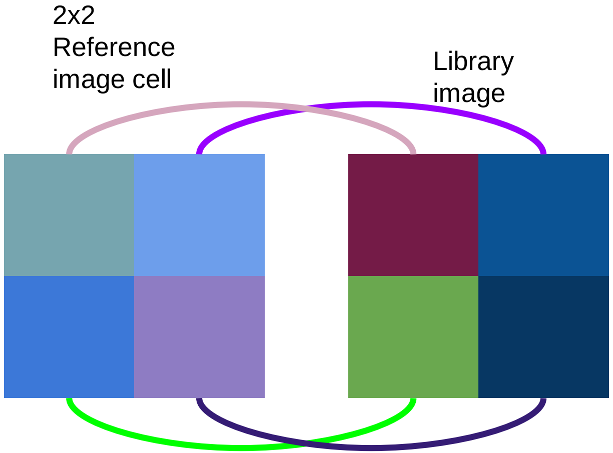


Real Photomosaic



Advanced Photomosaic

- How it works
  - Divide reference images into cell images
  - Pixel-to-pixel matching
  - Construct with matching images
- More computationally expensive



Advanced Photomosaic