Image Filter

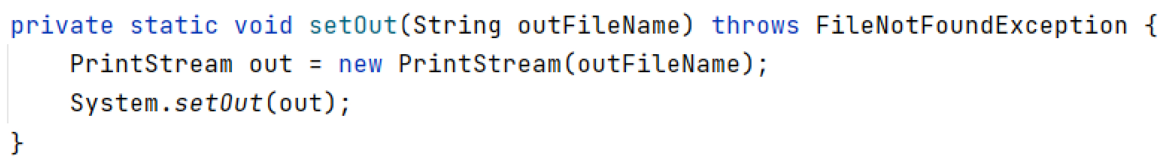
Code structure:

There are 5 major classes used in the application:

- **TestImageFilter** - the application class that starts all the filters in the correct order and performs speed up calculations and images verification.
- **IOImageTool** - the helper class that performs image I/O operations.
- **ImageFilter** - *iterative nine-point image convolution filter working on linearized image. In each of the NRSTEPS iteration steps, the average RGB-value of each pixel in the source array is computed taking into account the pixel and its 8 neighbor pixels (in 2D) and written to the destination array. (Assignment1 class description)*
- **ParallelFJImageFilter** - parallel filter that uses the same filtering logic as *ImageFilter* but using the Fork / Join pool principle.
- **ParallelFJImageFilterAction** - recursive action class that splits the image filtering into small parts and processes these parts.

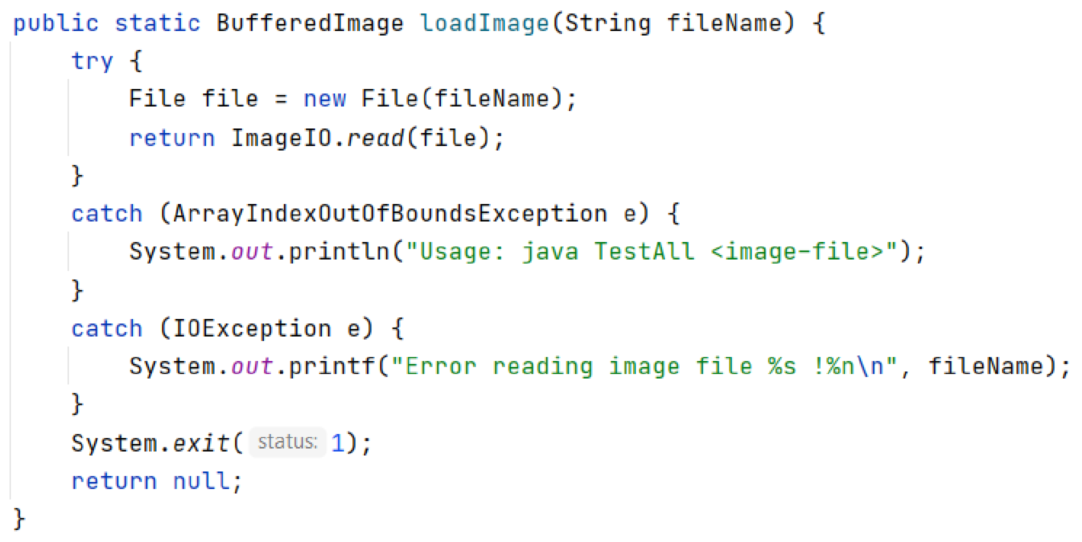
Execution:

The program execution starts with *main* method of the *TestImageFilter* class. The program takes two arguments: image file name (*args[0]*) and output file name (*args[1]*) - **java TestImageFilter IMAGE1.JPG out1.txt**. The system out is redirected to the file using method *setOut*:



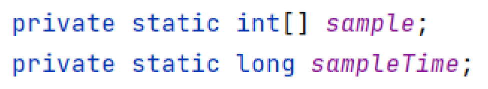
```
private static void setOut(String outFileName) throws FileNotFoundException {
    PrintStream out = new PrintStream(outFileName);
    System.setOut(out);
}
```

The image is loaded using the *loadImage* method of *IOImageTool* class:



```
public static BufferedImage loadImage(String fileName) {
    try {
        File file = new File(fileName);
        return ImageIO.read(file);
    }
    catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Usage: java TestAll <image-file>");
    }
    catch (IOException e) {
        System.out.printf("Error reading image file %s !%n\n", fileName);
    }
    System.exit( status: 1);
    return null;
}
```

First, the sequential filter is running. It's results: execution time, filtered image array – are saved as samples to *TestImageFilter* class fields:



```
private static int[] sample;
private static long sampleTime;
```

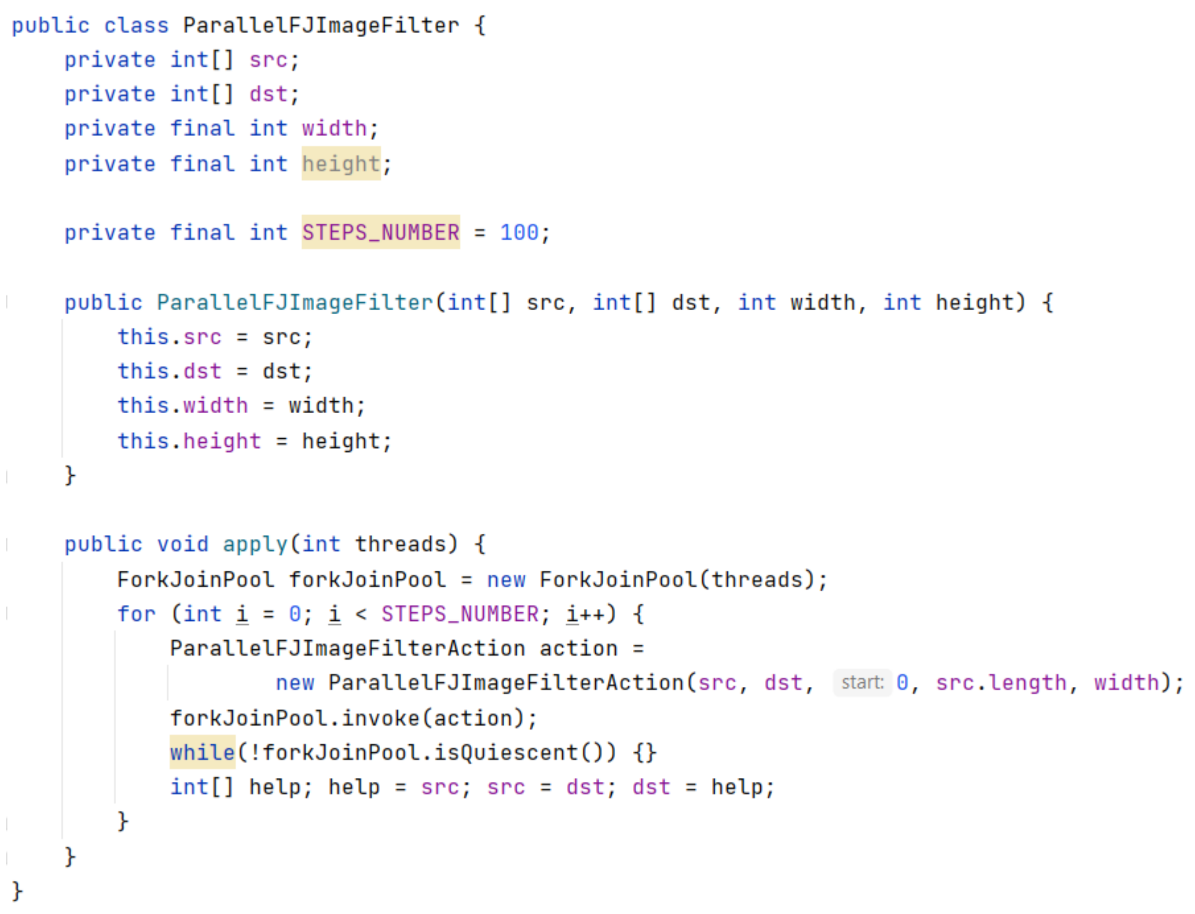
Then parallel filter is running using different number of threads (1, 2, 4, 8, 16, 32):

```java
for (int i = 0; i < THREADS.length; i++) {
    runParallelFilter(img, w, h, String.format("ParallelFiltered%d_%s", THREADS[i], srcFileName), THREADS[i]);
}
```

The *runParallelFilter* method invokes the ParallelFJImageFilter with given number of threads and records the execution time and resulting image. It also performs comparison of its results to results from sequential filter: calculates the speed up and validates the final image (pixel-wise comparison with the image from sequential filter) - and logs them to the output file.

```java
private static void runParallelFilter(BufferedImage image, int width, int height,
                                      String outFileName, int threadsNumber) throws IOException {
    int[] src = IOImageTool.getRGB(image, width, height);
    int[] dst = new int[src.length];

    System.out.printf("Starting parallel image filter using %d threads.\n", threadsNumber);
    long startTime = System.currentTimeMillis();
    ParallelFJImageFilter filter = new ParallelFJImageFilter(src, dst, width, height);
    filter.apply(threadsNumber);
    long endTime = System.currentTimeMillis();

    long t = endTime - startTime;
    double speedUp = sampleTime / (double) t;
    double sampleSpeedUp = INITIAL_SPEED_UP * threadsNumber;
    boolean ok = speedUp >= sampleSpeedUp;

    System.out.printf("Parallel image filter took %d milliseconds using %d threads.\n", t, threadsNumber);
    System.out.println(verifyResult(sample, dst) ? "Output image verified successfully!" : "INCORRECT IMAGE!");
    System.out.printf("Speedup: %.5f %s (%s %.1f)\n",
            speedUp, ok ? "ok" : "not ok", ok ? ">=" : "<", sampleSpeedUp);

    IOImageTool.saveImage(dst, width, height, outFileName);

    System.out.printf("Output image: %s\n\n", outFileName);
}
```

Parallel image filter implementation:

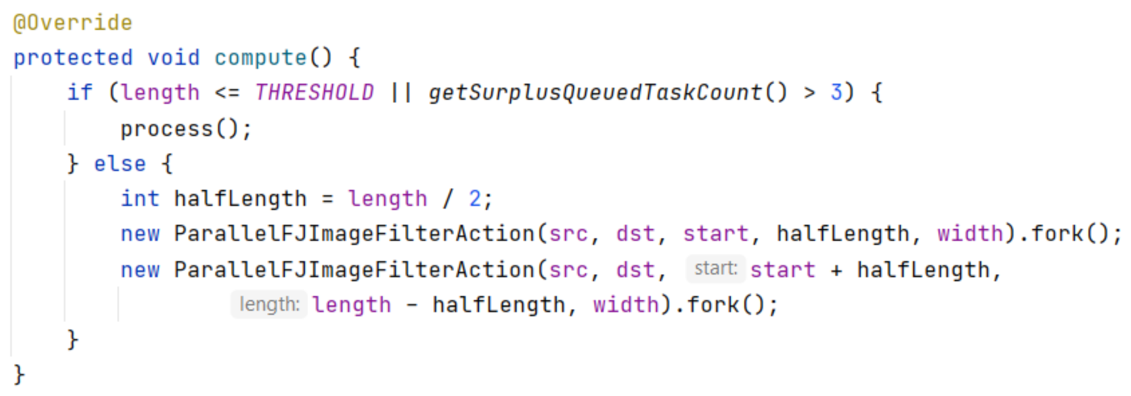


```java
public class ParallelFJImageFilter {
    private int[] src;
    private int[] dst;
    private final int width;
    private final int height;

    private final int STEPS_NUMBER = 100;

    public ParallelFJImageFilter(int[] src, int[] dst, int width, int height) {
        this.src = src;
        this.dst = dst;
        this.width = width;
        this.height = height;
    }

    public void apply(int threads) {
        ForkJoinPool forkJoinPool = new ForkJoinPool(threads);
        for (int i = 0; i < STEPS_NUMBER; i++) {
            ParallelFJImageFilterAction action =
                    new ParallelFJImageFilterAction(src, dst,  start: 0, src.length, width);
            forkJoinPool.invoke(action);
            while(!forkJoinPool.isQuiescent()) {}
            int[] help; help = src; src = dst; dst = help;
        }
    }
}
```

The *ParallelFJImageFilter* class iteratively performs parallel image filtering (*STEPS_NUMBER* steps) using *ForkJoinPool*. The synchronization happens after each iteration using method *isQuiescent*. This method returns *true* when all threads are idle, which means that they no longer can steal tasks from other queues and don't have tasks left for themselves.

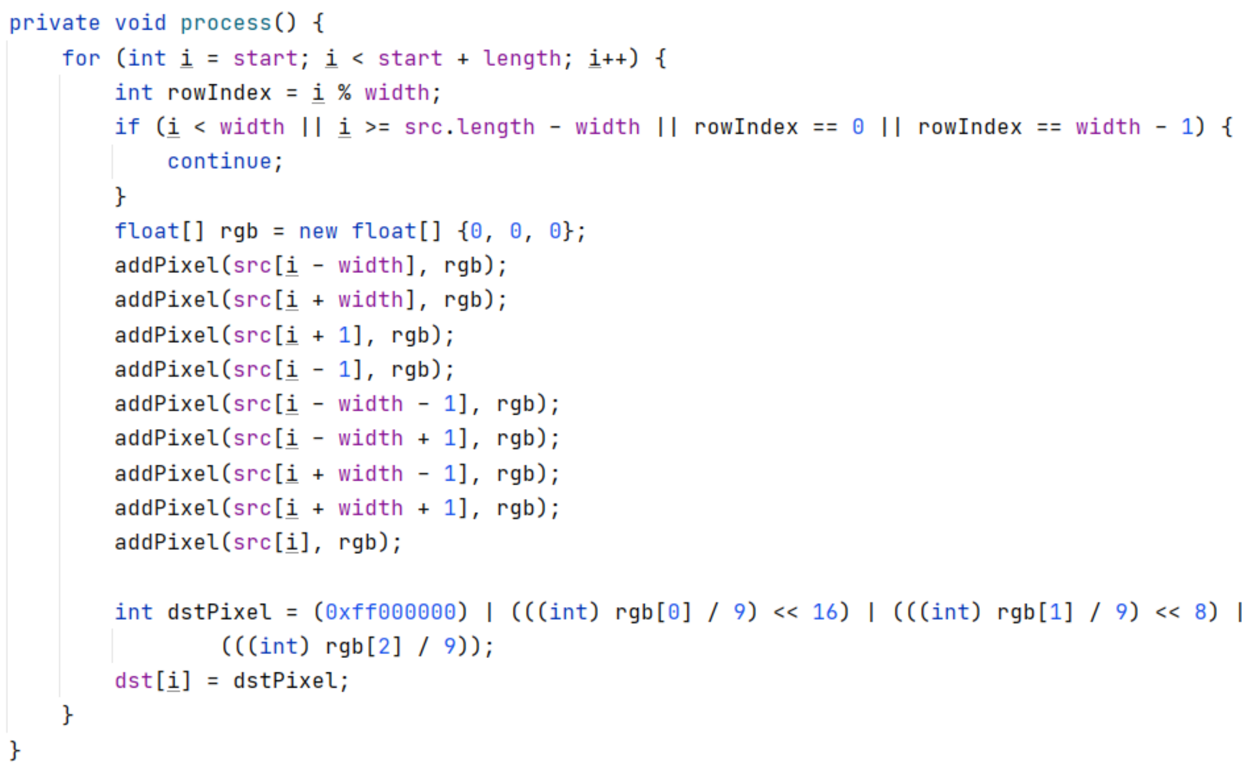
The *ParallelFJImageFilterAction* class performs actual image filtering. It extends *RecursiveAction* class which is a fork/join task. It overrides method *compute* – which contains the main computation performed by the class.

Here the task is rather divided into smaller parts or performed directly if the threshold values are met. The value of 100000 pixels (performed the best as a constant with respect to different thread numbers) is used as threshold in combination with dynamic threshold. The method *getSurplusQueuedTaskCount* is used to determine *an estimate of how many more locally queued tasks are held by the current worker thread than there are other worker threads that might steal them (Oracle documentation).* It is recommended to keep this a small constant value (in our case 3). If the threshold is not met the image array is divided into 2 parts.



```java
@Override
protected void compute() {
    if (length <= THRESHOLD || getSurplusQueuedTaskCount() > 3) {
        process();
    } else {
        int halfLength = length / 2;
        new ParallelFJImageFilterAction(src, dst, start, halfLength, width).fork();
        new ParallelFJImageFilterAction(src, dst,  start: start + halfLength,
                 length: length - halfLength, width).fork();
    }
}
```

Method *process* goes through the given part of the image array (the part from *start* index with specified *length*) and calculates average value for each pixel.



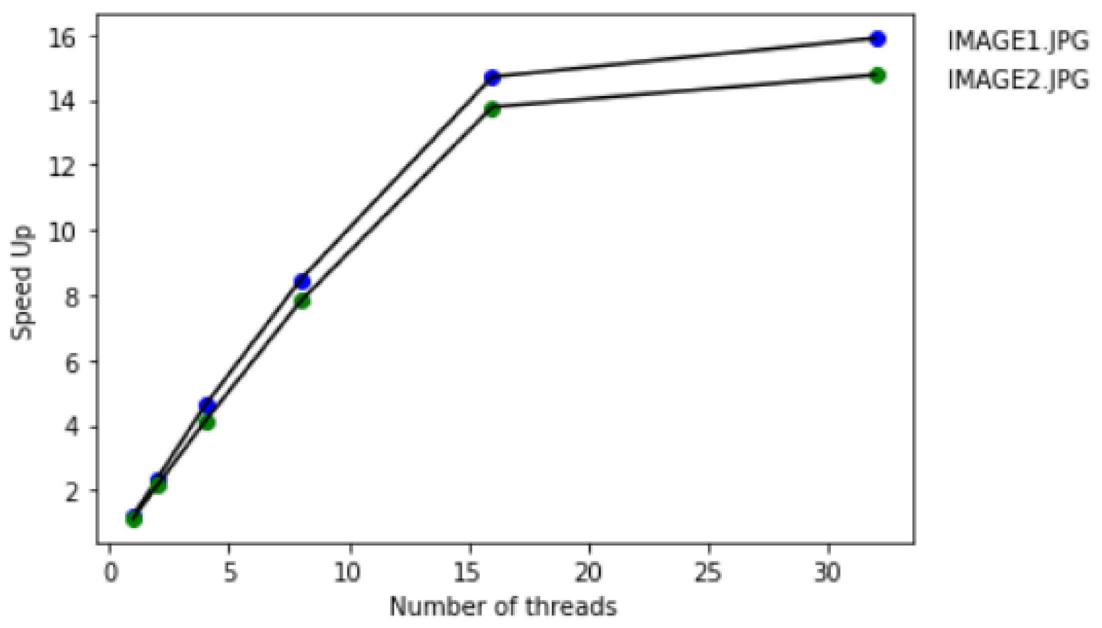
```java
private void process() {
    for (int i = start; i < start + length; i++) {
        int rowIndex = i % width;
        if (i < width || i >= src.length - width || rowIndex == 0 || rowIndex == width - 1) {
            continue;
        }
        float[] rgb = new float[] {0, 0, 0};
        addPixel(src[i - width], rgb);
        addPixel(src[i + width], rgb);
        addPixel(src[i + 1], rgb);
        addPixel(src[i - 1], rgb);
        addPixel(src[i - width - 1], rgb);
        addPixel(src[i - width + 1], rgb);
        addPixel(src[i + width - 1], rgb);
        addPixel(src[i + width + 1], rgb);
        addPixel(src[i], rgb);

        int dstPixel = (0xff000000) | (((int) rgb[0] / 9) << 16) | (((int) rgb[1] / 9) << 8) |
                (((int) rgb[2] / 9));
        dst[i] = dstPixel;
    }
}
```

The indexes of neighbor pixel indexes are calculated as follows:

i-w-1	i-w	i-w+1		
i-1	i	i+1		
i+w-1	i+w	i+w+1		

Results:

The following chart shows the speed up when using parallel image filter using different number of threads:



Here we can see that IMAGE1.JPG has better speed up when number of threads increases. This is due to the fact that the second picture is larger in size and requires more tasks to be processed. Nevertheless, the parallel filter for both pictures performed much better than the sequential filter and met the requirements for all numbers of threads (except 32 threads).