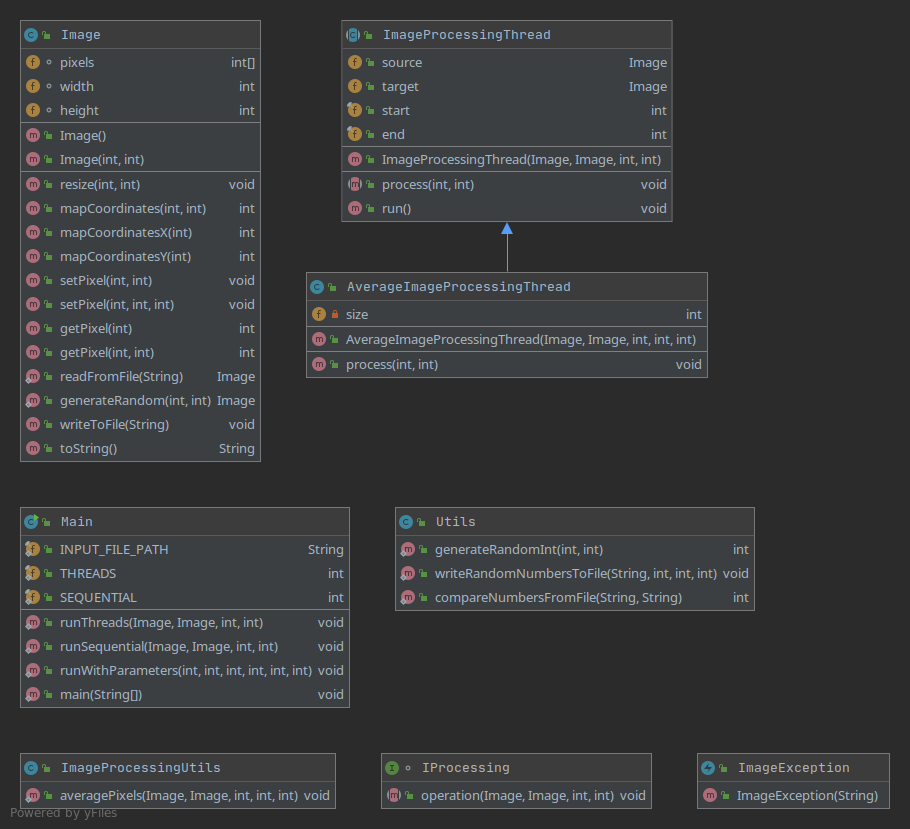
Laborator 1 - Tanislav Cosmin-Gabriel, Grup 237

**Implementation details**



First, I choose a running method based on the parameters being given: width, height, kernel size, sequential / threaded (with number of threads) and number of runs to average.

I use the Image::generateRandom function to generate a random image, then write it to a file, and read it again (to follow the instructions in the lab task).

After that, I run a sequential processing function or a threaded processing function.

The sequential processing function goes over all the pixels in the image and runs the ImageProcessingUtils::averagePixels function.

The threaded function splits the Image (linearly) into chunks of width \* height / number of threads pixels, accounting for the case when the Image doesn’t split evenly between all the threads, by extending the chunk of pixel by 1 for each thread until the rest is 0.

See below pseudo-code for explanation.

n = width \* height

chunk = n / number of threads

rest = n % number of threads

end = 0

for (i = 0; i < p; i++) {

start = end;

end = start + chunk;

if (end > n) {

end = n;

}

if (rest > 0) {

end += 1;

rest--;

}

[... do stuff with start and end ...]

}

A thread is spawned to process the pixels between every start and end, using the same function as the sequential implementation, ImageProcessingUtils::averagePixels.

The ImageProcessingUtils::averagePixels function will average all pixels in the kernel size specified, with the target pixel being the furthest bottom-right one.

The resulting pixel will be placed in the target image so as to not modify the source image and scramble the results of other subsequent pixel processing.

The Image class abstracts away the storage and provides methods for getting the pixel value for both linear and 2D indices. Internally, the pixels are stored linearly and the following formulas are used for translating between the two representation types.

int mapCoordinates(int x, int y) {

return y \* width + x;

}

int mapCoordinatesX(int i) {

return i % width;

}

int mapCoordinatesY(int i) {

return i / width;

}

**Results**

|  |  |
| --- | --- |
| Type | Average thread time |
| N=M=10 si n=m=3 |  |
| Java, p=4 | 0ms |
| Java, p=sequential | 0ms |
| C++, p=4 | 0ms |
| C++, p=sequential | 0ms |
|  |  |
| N=M=1000 si n=m=5 |  |
| Java, p=sequential | 48ms |
| Java, p=2 | 39ms |
| Java, p=4 | 20ms |
| Java, p=8 | 21ms |
| Java, p=16 | 19ms |
| C++, p=sequential | 66ms |
| C++, p=2 | 28ms |
| C++, p=4 | 16ms |
| C++, p=8 | 15ms |
| C++, p=16 | 15ms |
|  |  |
| N=10 M=10000 si n=m=5 |  |
| Java, p=sequential | 5ms |
| Java, p=2 | 2ms |
| Java, p=4 | 1ms |
| Java, p=8 | 2ms |
| Java, p=16 | 3ms |
| C++, p=sequential | 6ms |
| C++, p=2 | 2ms |
| C++, p=4 | 1ms |
| C++, p=8 | 1ms |
| C++, p=16 | 1ms |
|  |  |
| N=10000 M=10 si n=m=5 |  |
| Java, p=sequential | 4ms |
| Java, p=2 | 3ms |
| Java, p=4 | 2ms |
| Java, p=8 | 3ms |
| Java, p=16 | 2ms |
| C++, p=sequential | 5ms |
| C++, p=2 | 2ms |
| C++, p=4 | 1ms |
| C++, p=8 | 1ms |
| C++, p=16 | 1ms |

**Conclusions**

The Java implementation behaves a bit better sequentially, while C++ (ran in release mode) behaves better across all multi-threaded tests, probably because of the lower latency of spawning a thread.

In C++, using a static array of a predefined size didn’t bring any performance benefits, as I was already reserving width \* height bytes in the dynamic size implementation to avoid re-allocations.