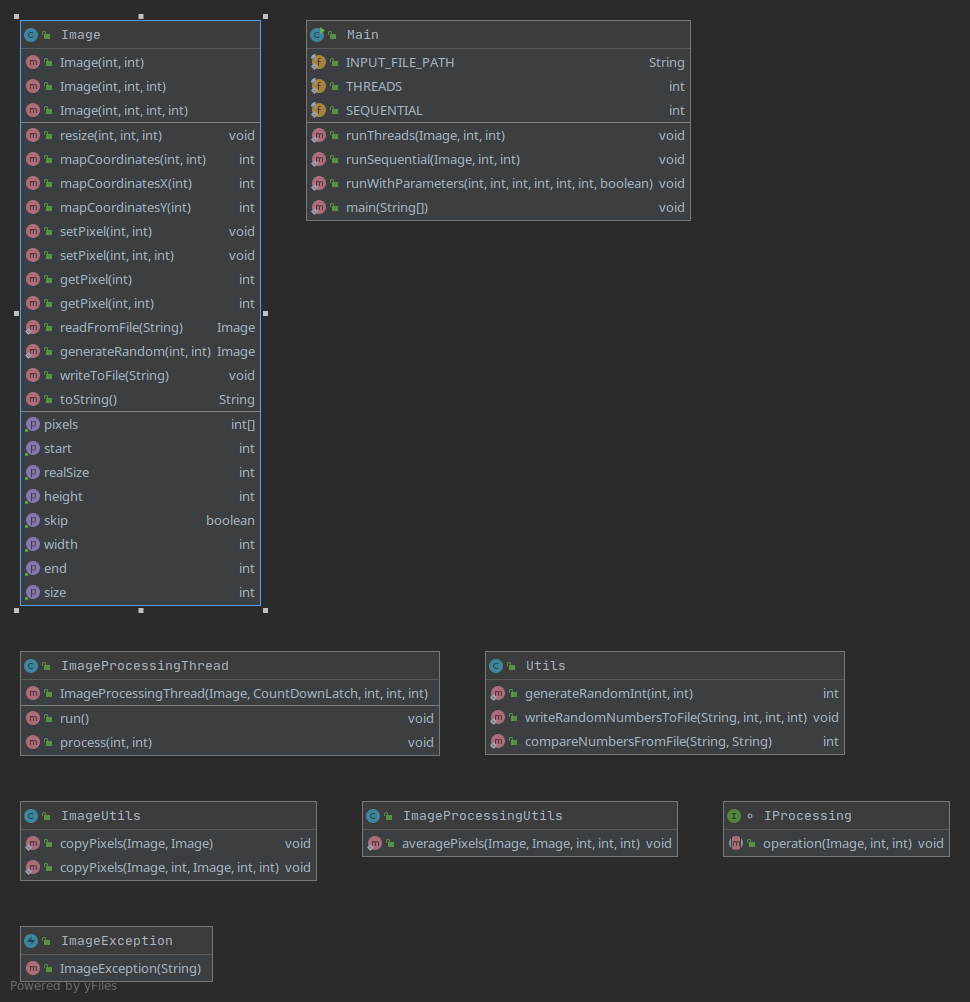
Laborator 2 - Tanislav Cosmin-Gabriel, Grup 237

**Implementation details**



**Program flow**

The program begins by iterating over all the tests to be ran and timed. If the width and height of the image needed by the test changed, it generates a new random image matching the requested width and height. It uses Image::generateRandom function to generate a random image, then writes it to a file.

A processing function is chosen based on whether the test is threaded or sequential. The processing function is ran.

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 ...]

}

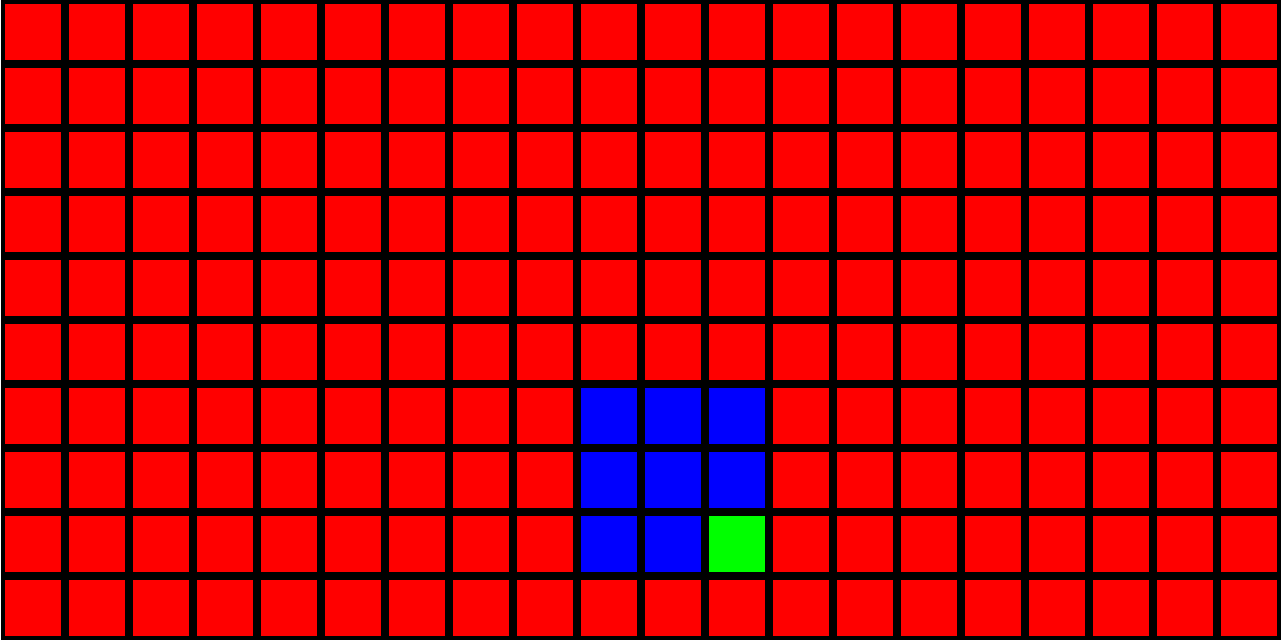
A thread is spawned to process the pixels between every start and end, using the same function as the sequential implementation, ImageProcessingUtils::averagePixels.

The thread creates a “skipped image” that virtually behaves the same as a basic image, but only stores data between a given start and end parameter. This skipped image is used as a target for the pixel averaging operation so as to not damage the data inside the source image. A count down latch is used to wait for all the threads to process their data, so that it can be written back into the source image, making the filter application basically in-place.

**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.

See the image below for explanation on a kernel size of 3.



blue = area over which the filter is ran

green = the target pixel

red = other pixels

**Image**

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;

}

The Image class also implements “skipped images” by receiving a start and an end position, and subtracting the start position from the linear index on getPixel and setPixel methods.

The given linear index is also checked to be between start and end, so as not to access data outside of the allocated array.

For a basic image, the start is 0 and the end is the real size of the image, eliminating the need for any special logic.

**Results**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Type | Threading | Time in ms | | | | | |
| C++ | C++ \*) | C++ \*\*) | Java | Java \*) | Java \*\*) |
| N=M=10  n=m=3 | sequential | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 threads | 0 | 0 | 0 | 0 | 5 | 7 |
| N=M=1000  n=m=5 | sequential | 66 | 38 | 38 | 48 | 50 | 79 |
| 2 threads | 28 | 29 | 33 | 39 | 34 | 47 |
| 4 threads | 16 | 16 | 18 | 19 | 17 | 28 |
| 8 threads | 15 | 16 | 21 | 21 | 18 | 28 |
| 16 threads | 15 | 18 | 24 | 19 | 20 | 29 |
| N=10000 M=10  n=m=5 | sequential | 6 | 3 | 3 | 5 | 3 | 6 |
| 2 threads | 2 | 2 | 3 | 2 | 3 | 4 |
| 4 threads | 1 | 2 | 2 | 1 | 2 | 3 |
| 8 threads | 1 | 2 | 2 | 2 | 2 | 4 |
| 16 threads | 1 | 2 | 2 | 3 | 4 | 6 |
| N=10 M=10000  n=m=5 | sequential | 5 | 3 | 3 | 4 | 3 | 6 |
| 2 threads | 2 | 2 | 3 | 3 | 3 | 4 |
| 4 threads | 1 | 2 | 1 | 2 | 2 | 3 |
| 8 threads | 1 | 2 | 2 | 3 | 3 | 5 |
| 16 threads | 1 | 2 | 2 | 2 | 4 | 4 |

**\*)** input values read from the main image directly, output values written to separate skip image and copied back into the main image

**\*\*)** input values copied from the main image into skip image, and output values written to the main image directly

**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.