Image Processing using TBB

**Author**: Matthew Fowler

**Student ID**: 17025958

**Module**: Parallel and Concurrent Programming

**Module Code**: CS3S666

**Module lecturers**: Marius Miknis, Simon Payne

# Speed Tests

**System specifications**: I am running an AMD Ryzen 5 2600 (3.40 GHz) with Windows 10.

I have used the “chrono” library to measure the time taken to execute specific sections of code. The main function contains adjustable variables to control how many times each solution should be executed. The average time to execute a solution through these iterations is automatically calculated and displayed.

When executing the program, most of the time is spent loading or saving files to the disk. I had initially overlooked this when setting up the speed tests, meaning that it was often hard to notice any significant speed-up or slow-down between the sequential and parallel solutions. As such, the speed tests only encapsulate the sections of code that are different in each solution.

Another significant factor for parallel solutions that use blocked\_range is the time taken to call range.begin() and range.end(). In previous version of the program, the for loops directly called these methods each iteration. This caused the parallel solution for task 3 to be slower than the sequential. For this reason, each time blocked\_range is used the program only calls these methods once and saves the result.

## Task 1: combining images

As shown in Figure 1 and Figure 2, the average time taken to execute all parts of the sequential solution is 447.6ms, compared to 97.4ms for the parallel solution. This means that the parallel solution is on average 4.6 times faster. This would be significant if the number of items to iterate over was larger, or if most of the processing time was spent on combining the images.

In this case however, it only saves 0.35 seconds. Considering the time taken to save the image is around 1.5s (see Figure 4), there is little benefit to parallelising this solution.

## Task 2: blur and threshold

The average time taken to execute the sequential Gaussian blur was close to 60 seconds on average, compared to 8.7 seconds for the parallel equivalent. This is a speed increase of 6.8 times.

In this situation the kernel is a 3x3 matrix, meaning that for each pixel in the output, 9 must be sampled from the input. For a 5000x7000 image, this means 315 million iterations. In these situations, parallelising the solution has a significant and noticeable impact.

## Task 3: number of white pixels and mask inversion

The time taken to execute the sequential solution was 159.4ms on average, compared to just 19.2ms for the parallel solution. The relative speed increase is roughly the same as the solutions to task 1, with the parallel solution being 3.80 times faster than the sequential.

When initially testing this solution, I did not include an adjustable grain size for the parallel\_for and parallel\_reduce functions. Figure 7 shows the results of these tests. The speeds for individual tests varied significantly, with the mask filter section ranging from 13ms to 31ms. This was due to the parallel\_for function dynamically calculating the grain size at run-time, with the result varying depending on the current resources available to it. Through trial and error, I was able to find that a grain size of 32768 gave consistently faster results (I arrived at this by starting with a value of 128 and doubling the value until the tests ran slower than the previous).

# Appendix A – Images & Screenshots

# 

Figure 1: speed tests for sequential solutions

# 

Figure 2: Speed tests for parallel solutions

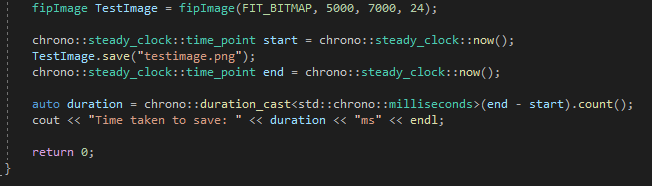


Figure 3: speed test for saving a 5000x7000 pixel bitmap image

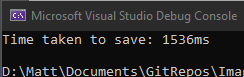


Figure 4: result of executing figure 3

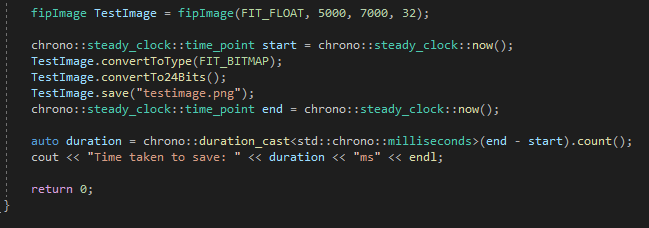


Figure 5: speed test for both converting a 32 bpp image to a 24 bpp image AND then saving it

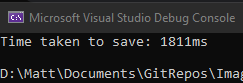


Figure 6: result of executing figure 5

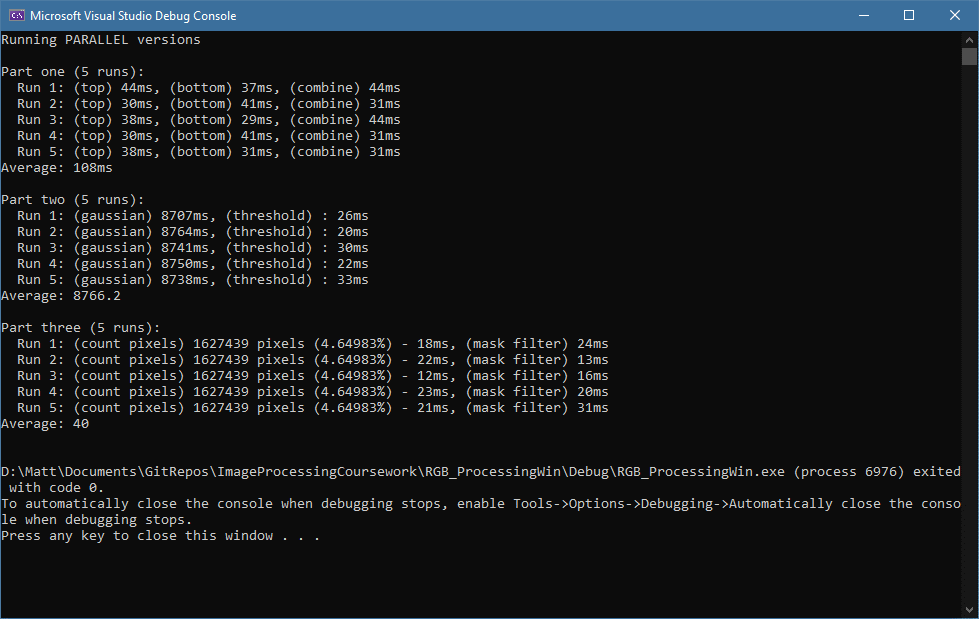


Figure : speed tests for parallel solution without setting the grain size

# Appendix B – Code

#include <iostream>

#include <vector>

//Thread building blocks library

#include <tbb/task\_scheduler\_init.h>

//Free Image library

#include <FreeImagePlus.h>

//My includes

#include <functional>

#include <thread>

#include <chrono>

#include <cmath>

#include <tbb/parallel\_for.h>

#include <tbb/blocked\_range.h>

#include <tbb/blocked\_range2d.h>

#include <tbb/parallel\_reduce.h>

//Defines

#define M\_PIf 3.14159265358979f // reduces computation overhead when using pi.

using namespace std;

using namespace tbb;

// -------------------- TYPES --------------------

struct pixel\_rgb

{

alignas(1) unsigned char r = 0;

alignas(1) unsigned char g = 0;

alignas(1) unsigned char b = 0;

};

// generic functor parent

class StencilTask

{

public:

virtual float operator()(float x, float y) = 0;

};

// Gaussian functor

// use constructor to set Gaussian parameters 'sigma' and 'kernal pre-load radius' e.g. GaussianBlur MyGaussian = GaussianBlur(0.8f, 2)

// once instantiated, object can be called as if it was a function e.g. MyGaussian(x\_coord, y\_coord)

// NOTE: 'kradius' is the number of pixels away from the origin that the mask is applied to, e.g. kradius=2 means a 5x5 matrix

class GaussianBlur : public StencilTask

{

private:

float sigma;

int kradius;

vector<vector<float>> kernel;

// Pre-load mask matrix to avoid computing the same value multiple times.

// With kradius=0 no values will be pre-loaded and everything is computed dynamically.

void loadKernel(int kradius)

{

// x and y start from -kradius and finish at kradius inclusively.

for (int x = -(kradius); x <= kradius; x++)

{

vector<float> curRow;

for (int y = -(kradius); y <= kradius; y++)

{

curRow.push\_back(1.0f / (2.0f \* M\_PIf \* pow(sigma, 2)) \* exp(-((pow(x, 2) + pow(y, 2)) / (2.0f \* pow(sigma, 2.0f)))));

}

kernel.push\_back(curRow);

}

}

public:

// 'kradius' is the number of values to the left and right of the origin in the pre-loaded mask matrix.

// It is not a hard limit on the number of computable values.

GaussianBlur(float sigma, int kradius = 0)

{

this->sigma = sigma;

this->kradius = kradius;

this->loadKernel(kradius);

}

float operator()(float x, float y)

{

if (abs(x) <= kradius && abs(y) <= kradius) return kernel[x + kradius][y + kradius]; //value HAS been pre-loaded

else return 1.0f / (2.0f \* M\_PIf \* pow(sigma, 2)) \* exp(-((pow(x, 2) + pow(y, 2)) / (2.0f \* pow(sigma, 2)))); //value has NOT been pre-loaded

}

};

// A component of ComineImagesParallel. Combines only a sub-section of the given images.

void CombineSubImageAnd(pixel\_rgb\* aPointer, pixel\_rgb\* bPointer, pixel\_rgb\* outPointer, unsigned int numIterations)

{

// White and black pixel definitions

pixel\_rgb WHITE\_PIXEL;

pixel\_rgb BLACK\_PIXEL;

WHITE\_PIXEL.r = 255; WHITE\_PIXEL.g = 255; WHITE\_PIXEL.b = 255;

BLACK\_PIXEL.r = 0; BLACK\_PIXEL.g = 0; BLACK\_PIXEL.b = 0;

for (unsigned int pixel = 0; pixel < numIterations; pixel++, aPointer++, bPointer++, outPointer++)

{

if (aPointer->r == bPointer->r

&& aPointer->g == bPointer->g

&& aPointer->b == bPointer->b)

{

\*aPointer = BLACK\_PIXEL;

}

else

{

\*aPointer = WHITE\_PIXEL;

}

}

}

// A component of ComineImagesParallel. Combines only a sub-section of the given images.

void CombineSubImageSum(pixel\_rgb\* aPointer, pixel\_rgb\* bPointer, pixel\_rgb\* outPointer, unsigned int numIterations)

{

for (unsigned int pixel = 0; pixel < numIterations; pixel++, aPointer++, bPointer++, outPointer++)

{

aPointer->r = (aPointer->r / (unsigned char)2) + (bPointer->r / (unsigned char)2);

aPointer->g = (aPointer->g / (unsigned char)2) + (bPointer->g / (unsigned char)2);

aPointer->b = (aPointer->b / (unsigned char)2) + (bPointer->b / (unsigned char)2);

}

}

int main()

{

int nt = task\_scheduler\_init::default\_num\_threads();

task\_scheduler\_init T(nt);

// Image file paths

char IN\_TOP\_1[] = "../Images/render\_top\_1.png";

char IN\_TOP\_2[] = "../Images/render\_top\_2.png";

char IN\_BOTTOM\_1[] = "../Images/render\_bottom\_1.png";

char IN\_BOTTOM\_2[] = "../Images/render\_bottom\_2.png";

char OUT\_STAGE\_1\_TOP[] = "../Images/stage1\_top.png";

char OUT\_STAGE\_1\_BOTTOM[] = "../Images/stage1\_bottom.png";

char OUT\_STAGE\_1\_COMBINED[] = "../Images/stage1\_combined.png";

char OUT\_STAGE\_2\_BLURRED[] = "../Images/stage2\_blurred.png";

char OUT\_STAGE\_2\_THRESHOLD[] = "../Images/stage2\_threshold.png";

char OUT\_STAGE\_3[] = "../Images/stage3\_final.png";

// Number of times each stage will be executed

const bool DO\_PARALLEL = true;

const int STAGE\_1\_ITERATIONS = 1;

const int STAGE\_2\_ITERATIONS = 1;

const int STAGE\_3\_ITERATIONS = 1;

std::chrono::steady\_clock::time\_point start;

std::chrono::steady\_clock::time\_point end;

float average = 0.0f;

// Grain size for parallel\_for and parallel\_reduce:

const int GRAIN\_SIZE = 36768;

// FIP image objects:

fipImage imgInputA;

fipImage imgInputB;

fipImage imgOutput;

// White and black pixel definitions

pixel\_rgb WHITE\_PIXEL;

pixel\_rgb BLACK\_PIXEL;

WHITE\_PIXEL.r = 255; WHITE\_PIXEL.g = 255; WHITE\_PIXEL.b = 255;

BLACK\_PIXEL.r = 0; BLACK\_PIXEL.g = 0; BLACK\_PIXEL.b = 0;

// Whether or not images should be saved

const bool SAVE\_IMAGES = true;

if (DO\_PARALLEL) cout << "Running PARALLEL versions" << endl << endl;

else cout << "Running SEQUENTIAL versions" << endl << endl;

//Part 1 (Image Comparison): -----------------DO NOT REMOVE THIS COMMENT----------------------------//

cout << "Part one (" << STAGE\_1\_ITERATIONS << " runs):" << endl;

for (int i = 0; i < STAGE\_1\_ITERATIONS; i++)

{

// COMBINE TOP

// Load images using Free Image Plus library

imgInputA.load(IN\_TOP\_1);

imgInputB.load(IN\_TOP\_2);

imgInputA.convertTo24Bits();

imgInputB.convertTo24Bits();

// Image dimensions

unsigned int numPixels = imgInputA.getWidth() \* imgInputA.getHeight();

// Iterate over each pixel

pixel\_rgb\* aPointer = (pixel\_rgb\*)imgInputA.accessPixels(); // also acts as the output image

pixel\_rgb\* bPointer = (pixel\_rgb\*)imgInputB.accessPixels();

start = std::chrono::steady\_clock::now();

if (DO\_PARALLEL == false)

{

// SEQUENTIAL SOLUTION

for (unsigned int pixel = 0; pixel < numPixels; pixel++, aPointer++, bPointer++)

{

if (aPointer->r == bPointer->r

&& aPointer->g == bPointer->g

&& aPointer->b == bPointer->b)

{

\*aPointer = BLACK\_PIXEL;

}

else

{

\*aPointer = WHITE\_PIXEL;

}

}

}

if (DO\_PARALLEL)

{

// PARALLEL SOLUTION

// Create threads to process smaller sub-images

vector<thread> threads;

unsigned int stepsize = numPixels / nt;

unsigned int remainder = numPixels % nt;

for (int i = 0; i < nt; i++)

{

if (i == 0)

{

threads.push\_back(thread(CombineSubImageAnd, aPointer, bPointer, aPointer, stepsize + remainder));

aPointer += stepsize + remainder;

bPointer += stepsize + remainder;

}

else

{

threads.push\_back(thread(CombineSubImageAnd, aPointer, bPointer, aPointer, stepsize));

aPointer += stepsize;

bPointer += stepsize;

}

}

// Wait for the threads to finish executing

for (auto& thread : threads)

{

thread.join();

}

}

end = std::chrono::steady\_clock::now();

// Save output image to disk

if (SAVE\_IMAGES) imgInputA.save(OUT\_STAGE\_1\_TOP);

auto duration\_p1\_s1 = chrono::duration\_cast<std::chrono::milliseconds>(end - start).count();

cout << " Run " << i + 1 << ": (top) " << duration\_p1\_s1 << "ms, ";

average += (float)duration\_p1\_s1;

// COMBINE BOTTOM

// Load images using Free Image Plus library

imgInputA.load(IN\_BOTTOM\_1);

imgInputB.load(IN\_BOTTOM\_2);

imgInputA.convertTo24Bits();

imgInputB.convertTo24Bits();

// Iterate over each pixel

aPointer = (pixel\_rgb\*)imgInputA.accessPixels(); // also acts as the output image

bPointer = (pixel\_rgb\*)imgInputB.accessPixels();

start = std::chrono::steady\_clock::now();

if (DO\_PARALLEL == false)

{

// SEQUENTIAL SOLUTION

for (unsigned int pixel = 0; pixel < numPixels; pixel++, aPointer++, bPointer++)

{

if (aPointer->r == bPointer->r

&& aPointer->g == bPointer->g

&& aPointer->b == bPointer->b)

{

\*aPointer = BLACK\_PIXEL;

}

else

{

\*aPointer = WHITE\_PIXEL;

}

}

}

if (DO\_PARALLEL)

{

// PARALLEL SOLUTION

// Create threads to process smaller sub-images

vector<thread> threads;

unsigned int stepsize = numPixels / nt;

unsigned int remainder = numPixels % nt;

for (int i = 0; i < nt; i++)

{

if (i == 0)

{

threads.push\_back(thread(CombineSubImageAnd, aPointer, bPointer, aPointer, stepsize + remainder));

aPointer += stepsize + remainder;

bPointer += stepsize + remainder;

}

else

{

threads.push\_back(thread(CombineSubImageAnd, aPointer, bPointer, aPointer, stepsize));

aPointer += stepsize;

bPointer += stepsize;

}

}

// Wait for the threads to finish executing

for (auto& thread : threads)

{

thread.join();

}

}

end = std::chrono::steady\_clock::now();

// Save output image to disk

if (SAVE\_IMAGES) imgInputA.save(OUT\_STAGE\_1\_BOTTOM);

auto duration\_p1\_s2 = chrono::duration\_cast<std::chrono::milliseconds>(end - start).count();

cout << "(bottom) " << duration\_p1\_s2 << "ms, ";

average += (float)duration\_p1\_s2;

// COMBINE TOP AND BOTTOM

// Load images using Free Image Plus library

imgInputA.load(OUT\_STAGE\_1\_TOP);

imgInputB.load(OUT\_STAGE\_1\_BOTTOM);

imgInputA.convertTo24Bits();

imgInputB.convertTo24Bits();

// Iterate over each pixel

aPointer = (pixel\_rgb\*)imgInputA.accessPixels(); // also acts as the output image

bPointer = (pixel\_rgb\*)imgInputB.accessPixels();

start = std::chrono::steady\_clock::now();

if (DO\_PARALLEL == false)

{

// SEQUENTIAL SOLUTION

for (unsigned int pixel = 0; pixel < numPixels; pixel++, aPointer++, bPointer++)

{

aPointer->r = (aPointer->r / (unsigned char)2) + (bPointer->r / (unsigned char)2);

aPointer->g = (aPointer->g / (unsigned char)2) + (bPointer->g / (unsigned char)2);

aPointer->b = (aPointer->b / (unsigned char)2) + (bPointer->b / (unsigned char)2);

}

}

if (DO\_PARALLEL)

{

// PARALLEL SOLUTION

// Create threads to process smaller sub-images

vector<thread> threads;

unsigned int stepsize = numPixels / nt;

unsigned int remainder = numPixels % nt;

for (int i = 0; i < nt; i++)

{

if (i == 0)

{

threads.push\_back(thread(CombineSubImageSum, aPointer, bPointer, aPointer, stepsize + remainder));

aPointer += stepsize + remainder;

bPointer += stepsize + remainder;

}

else

{

threads.push\_back(thread(CombineSubImageSum, aPointer, bPointer, aPointer, stepsize));

aPointer += stepsize;

bPointer += stepsize;

}

}

// Wait for the threads to finish executing

for (auto& thread : threads)

{

thread.join();

}

}

end = std::chrono::steady\_clock::now();

// Save output image to disk

if (SAVE\_IMAGES) imgInputA.save(OUT\_STAGE\_1\_COMBINED);

auto duration\_p1\_s3 = chrono::duration\_cast<std::chrono::milliseconds>(end - start).count();

cout << "(combine) " << duration\_p1\_s3 << "ms" << endl;

average += (float)duration\_p1\_s3;

}

if (STAGE\_1\_ITERATIONS > 0) average = average / (float)STAGE\_1\_ITERATIONS;

cout << "Average: " << average << "ms" << endl << endl;

average = 0.0f;

//Part 2 (Blur & post-processing): -----------DO NOT REMOVE THIS COMMENT----------------------------//

// blur parameters

float sigma = 0.8f;

int kernal\_radius = 1;

// Setup blur functor

GaussianBlur BlurFunc = GaussianBlur(sigma, kernal\_radius);

cout << "Part two (" << STAGE\_2\_ITERATIONS << " runs): " << endl;

for (int i = 0; i < STAGE\_2\_ITERATIONS; i++)

{

// GAUSSIAN BLUR

// Load input image from disk into memory

imgInputA.load(OUT\_STAGE\_1\_COMBINED);

imgInputA.convertToFloat();

// Load empty output image into memory

unsigned int width = imgInputA.getWidth();

unsigned int height = imgInputA.getHeight();

unsigned int numPixels = width \* height;

imgOutput = fipImage(FIT\_FLOAT, width, height, 32);

// Setup variables for accessing memory

float\* fInPointer = (float\*)imgInputA.accessPixels();

float\* fOutPointer = (float\*)imgOutput.accessPixels();

unsigned int stencilIndex = 0;

float sum = 0.0f;

start = std::chrono::steady\_clock::now();

if (DO\_PARALLEL == false)

{

for (int yOrigin = 0; yOrigin < height; yOrigin++)

{

for (int xOrigin = 0; xOrigin < width; xOrigin++, fOutPointer++)

{

sum = 0.0f;

for (int yStencil = -(kernal\_radius); yStencil <= kernal\_radius; yStencil++)

{

// Make sure that the stencil's y-coord is within range. If not, snap to the nearest border pixel.

int absoluteYStencil = yOrigin + yStencil;

if (absoluteYStencil < 0) absoluteYStencil = 0;

if (absoluteYStencil >= height) absoluteYStencil = height - 1;

for (int xStencil = -(kernal\_radius); xStencil <= kernal\_radius; xStencil++)

{

// Make sure that the stencil's x-coord is within range. If not, snap to the nearest border pixel.

int absoluteXStencil = xOrigin + xStencil;

if (absoluteXStencil < 0) absoluteXStencil = 0;

if (absoluteXStencil >= width) absoluteXStencil = width - 1;

stencilIndex = (absoluteYStencil \* width) + absoluteXStencil;

sum += fInPointer[stencilIndex] \* BlurFunc(xStencil, yStencil);

}

}

\*fOutPointer = sum;

}

}

}

if (DO\_PARALLEL)

{

parallel\_for(blocked\_range2d<int>(0, (int)height,

0, (int)width),

[&](const blocked\_range2d<int>& dim)

{

int xstart = dim.cols().begin();

int ystart = dim.rows().begin();

int xend = dim.cols().end();

int yend = dim.rows().end();

int xsum = 0;

int ysum = 0;

for (int x = xstart; x != xend; x++)

{

for (int y = ystart; y != yend; y++)

{

for (int kx = -kernal\_radius; kx <= kernal\_radius; kx++)

{

xsum = x + kx;

if (xsum < 0) xsum = 0;

if (xsum >= width) xsum = width - 1;

for (int ky = -kernal\_radius; ky <= kernal\_radius; ky++)

{

ysum = y + ky;

if (ysum < 0) ysum = 0;

if (ysum >= height) ysum = height - 1;

fOutPointer[(y \* width) + x] += fInPointer[(ysum \* width) + xsum] \* BlurFunc(kx, ky);

}

}

}

}

}

);

}

end = std::chrono::steady\_clock::now();

// Save output image to disk

imgOutput.convertToType(FREE\_IMAGE\_TYPE::FIT\_BITMAP);

imgOutput.convertTo24Bits();

if (SAVE\_IMAGES) imgOutput.save(OUT\_STAGE\_2\_BLURRED);

auto duration\_p2\_s1 = chrono::duration\_cast<std::chrono::milliseconds>(end - start).count();

cout << " Run " << i+1 << ": (gaussian) " << duration\_p2\_s1 << "ms, ";

// THRESHOLD (any non-black pixels become white)

// Load input images from disk as FreeImagePlus images

imgInputA.load(OUT\_STAGE\_2\_BLURRED);

imgInputA.convertTo24Bits();

// Iterate over each pixel and apply the given function

pixel\_rgb\* inPointer = (pixel\_rgb\*)imgInputA.accessPixels();

start = std::chrono::steady\_clock::now();

if (DO\_PARALLEL == false)

{

for (unsigned int pixel = 0; pixel < numPixels; pixel++, inPointer++)

{

if (inPointer->r != 0 || inPointer->g != 0 || inPointer->b != 0)

{

\*inPointer = WHITE\_PIXEL;

}

}

}

if (DO\_PARALLEL)

{

parallel\_for(blocked\_range<int>(0, (int)numPixels), [&](const blocked\_range<int>& range) {

int begin = range.begin();

int end = range.end();

for (int i = begin; i < end; i++)

{

if (inPointer[i].r != 0 || inPointer[i].g != 0 || inPointer[i].b != 0)

{

inPointer[i] = WHITE\_PIXEL;

}

}

});

}

end = std::chrono::steady\_clock::now();

// Save output image to disk

if (SAVE\_IMAGES) imgInputA.save(OUT\_STAGE\_2\_THRESHOLD);

auto duration\_p2\_s2 = chrono::duration\_cast<std::chrono::milliseconds>(end - start).count();

cout << "(threshold) " << ": " << duration\_p2\_s2 << "ms" << endl;

average += (float)duration\_p2\_s1 + (float)duration\_p2\_s2;

}

if (STAGE\_2\_ITERATIONS > 0) average = average / (float)STAGE\_2\_ITERATIONS;

cout << "Average: " << average << endl << endl;

average = 0.0f;

//Part 3 (Image Mask): -----------------------DO NOT REMOVE THIS COMMENT----------------------------//

// Part 3 sequential solution:

cout << "Part three (" << STAGE\_3\_ITERATIONS << " runs): " << endl;

for (int i = 0; i < STAGE\_3\_ITERATIONS; i++)

{

cout << " Run " << i+1 << ": (count pixels) ";

// NUMBER OF WHITE PIXELS

// Load input images from disk as FreeImagePlus images

imgInputB.load(OUT\_STAGE\_2\_THRESHOLD);

imgInputB.convertTo24Bits();

// Image dimensions

unsigned int numPixels = imgInputB.getWidth() \* imgInputB.getHeight();

// Iterate over each pixel and count how many pixels meet the given criteria

int sum = 0;

pixel\_rgb\* inPointer = (pixel\_rgb\*)imgInputB.accessPixels();

start = std::chrono::steady\_clock::now();

if (DO\_PARALLEL == false)

{

// SEQUENTIAL SOLUTION

for (unsigned int pixel = 0; pixel < numPixels; pixel++, inPointer++)

{

if (inPointer->r == 255

&& inPointer->g == 255

&& inPointer->b == 255)

{

sum++;

}

}

}

if (DO\_PARALLEL)

{

// PARALLEL SOLUTION

sum = parallel\_reduce(

blocked\_range<int>(0, numPixels),

0,

[&](const blocked\_range<int>& range, int initValue) {

int begin = range.begin();

int end = range.end();

for (int i = begin; i != end; i++)

{

if (inPointer[i].r == 255

&& inPointer[i].g == 255

&& inPointer[i].b == 255)

{

initValue++;

}

}

return initValue;

},

[&](int a, int b) {

return a + b;

}

);

}

end = std::chrono::steady\_clock::now();

cout << sum << " pixels (" << ((float)sum / (5000.0f \* 7000.0f)) \* 100.0f << "%) - ";

auto duration\_p3\_s1 = chrono::duration\_cast<std::chrono::milliseconds>(end - start).count();

cout << duration\_p3\_s1 << "ms, ";

// Load the input image and the mask from the disk

imgInputA.load(IN\_TOP\_1);

imgInputA.convertTo24Bits();

// For each pixel in the input, check that the corresponding pixel in the mask meets the condition

// If it does invert the pixel in the input image

inPointer = (pixel\_rgb\*)imgInputA.accessPixels();

pixel\_rgb\* maskPointer = (pixel\_rgb\*)imgInputB.accessPixels();

start = std::chrono::steady\_clock::now();

if (DO\_PARALLEL == false)

{

// SEQUENTIAL SOLUTION

for (unsigned int pixel = 0; pixel < numPixels; pixel++, inPointer++, maskPointer++)

{

if (maskPointer->r == 255

&& maskPointer->g == 255

&& maskPointer->b == 255)

{

inPointer->r = 255 - inPointer->r;

inPointer->g = 255 - inPointer->g;

inPointer->b = 255 - inPointer->b;

}

}

}

if (DO\_PARALLEL)

{

// PARALLEL SOLUTION

parallel\_for(blocked\_range<int>(0, numPixels), [&](blocked\_range<int>& range) {

int begin = range.begin();

int end = range.end();

for (int p = begin; p < end; p++)

{

if (maskPointer[p].r == 255

&& maskPointer[p].g == 255

&& maskPointer[p].b == 255)

{

inPointer[p].r = 255 - inPointer[p].r;

inPointer[p].g = 255 - inPointer[p].g;

inPointer[p].b = 255 - inPointer[p].b;

}

}

});

}

end = std::chrono::steady\_clock::now();

// Save result to the disk

if (SAVE\_IMAGES) imgInputA.save(OUT\_STAGE\_3);

auto duration\_p3\_s2 = chrono::duration\_cast<std::chrono::milliseconds>(end - start).count();

cout << "(mask filter) " << duration\_p3\_s2 << "ms" << endl;

average += (float)duration\_p3\_s1 + (float)duration\_p3\_s2;

}

if (STAGE\_3\_ITERATIONS > 0) average = average / (float)STAGE\_3\_ITERATIONS;

cout << "Average: " << average << endl << endl;

average = 0.0f;

return 0;

}