**Image Processing Using TBB Report**

Introduction

Within this report I will be looking at C++ Threading, Sequential and TBB implementation to compare their efficiency. This will be done by running and timing the speed for both sequential and TTB versions complete the task to check the difference in efficiency. The report will be split into 3 parts with part 1 looking at comparing and combining the pixels, Part 2 looks at applying a blur and threshold and part 3 calculating the white pixels percentage and applying a filter mask.

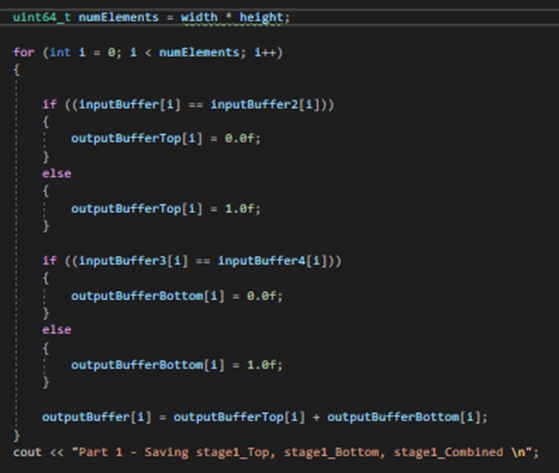
Part 1

In part 1 I will be loading the four images provided (pair 1: render\_top\_1 & render\_top\_2, pair 2: render\_bottom\_1 & render\_bottom\_2) and compare the pixels if the pixels are white, they will be made black and if they are not, they will be set to white. These are then saved as "stage1\_top.png" and "stage1\_bottom.png" which are then combined into a single image. There are 2 version provided sequential way and a second using C++11 multithreading code.

**Sequential**



First for the sequential version I start by loading each of the images one by one, to do this I first have to load the images by using .load(File Path) and then I converted it to a float using .convertToFloat. I then need to get the width and high of the inputimage using inputimage.getwidth and inputimage.getheight and save to auto width and auto height. This is so I can find number of pixels that are in the whole image. Then I created an input buffer for each image that allows me to access each of the individual pixels one by one using. accesspixels.



First, I needed to find the number of elements this was done by width \* height this was so that I knew the number of pixels when filtering though them. I then created a for loop that would filter though the pixels one by one until the it reached the last pixel. This was done by setting I = 0 and then checking if I was smaller than numElements and if it was increment I.

I then created an if statement that looks if the pixel in inputbuffer is the same as the pixel in inputbuffer2 then set the pixel to black, else set the pixel to white. The pixel was set to black by outputbuffertop[I] = 0.0f as 0.0f represents black and to set them to white I did outputbuffertop[I] = 1.0f as 1.0f represents white. This process was then repeated for the bottom pair.

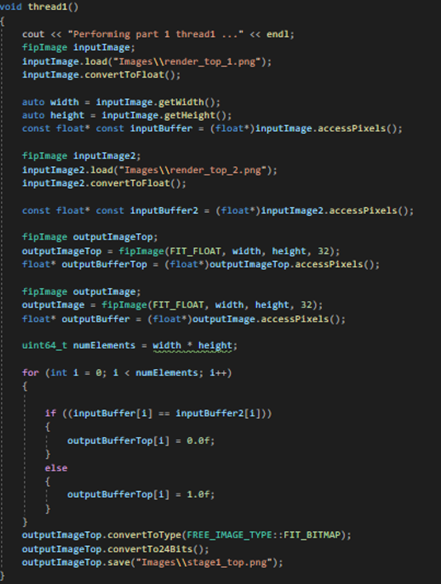
To combine the images, I set the outputbuffer to equal to the outputbuffertop + outputBufferBottom. This takes 50% of outputBufferTop pixels and 50% of outputBufferBottom pixels and set equal to them to create a combined imaging using both the top and the bottom.



Now that all the pixels have been compared and set to black or white and the top and bottom buffers have been combined, I have saved the images for stage1\_top, stage1\_bottom and stage1\_combined by first converting the image a bitmap. I then converted the images to 24bits using convertTo24Bits and lastly I saved all the images as .png images and named them “stage1\_top.png”, “stage1\_bottom.png” and “stage1\_combined.png”.

**C++11 multithreading code**

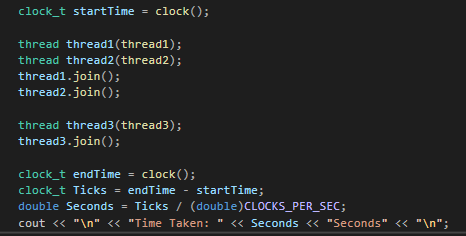
For the multithreading I separated the code out into individual parts thread1 containing top pixel comparison, thread2 containing the bottom pixel comparison and thread3 containing the combined imaged.



Here you can see both thread 1 and thread 2, they are very similar to the sequential version as it used the same code to load and compare the pixels. The difference is that the code is now split into separate threads with thread 1 containing the code to load the top pair and compare the pixel and thread 2 containing the bottom pair and comparing their pixels and saving them the same as the sequential way at the end of them both.



Thread 3 contains the combining of both images stage1\_top and stage1\_bottom to create one image stage1\_combined. It work by loading in stage1\_top.png and stage1\_bottom.png after they are created and sets the outputbuffer to equally the top inputbuffer plus the bottom inputbuffer. This works by adding the values of the pixels together.



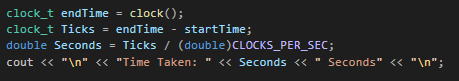
To run the multithreading, I first called the threads 1 and 2 and join them as this will run them at the same time and will create the pngs needed to run thread 3. I then call thread 3 and join that one to I join them using the .join command. The reason for using multi-threading is that it will load both the top and bottom at the same time and then it will combine the images instead of doing it one after the other this will help to improve the efficiency.

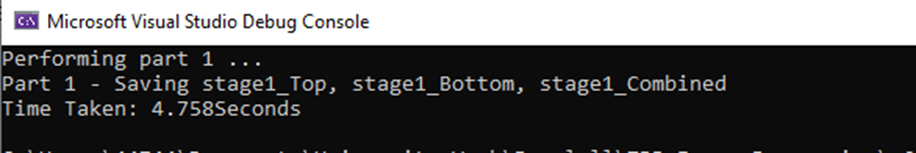
Part 1 Testing and Comparison

**Sequential**

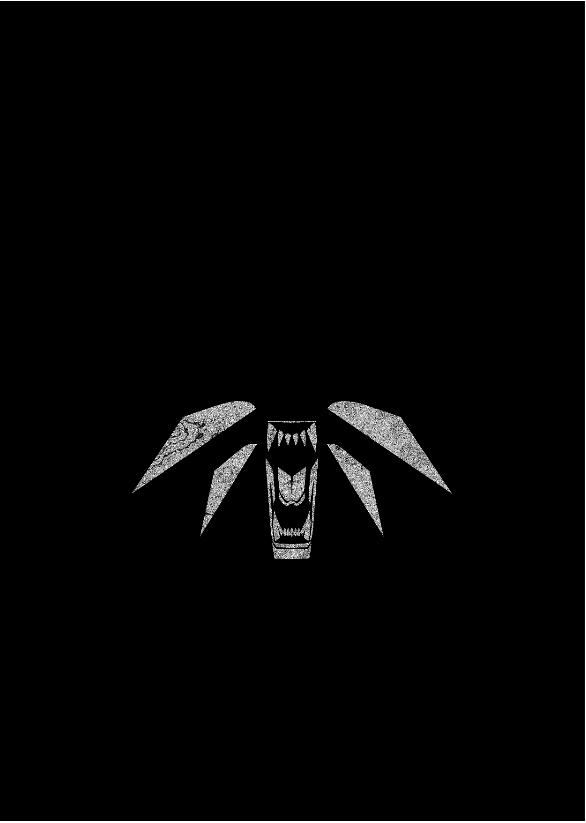
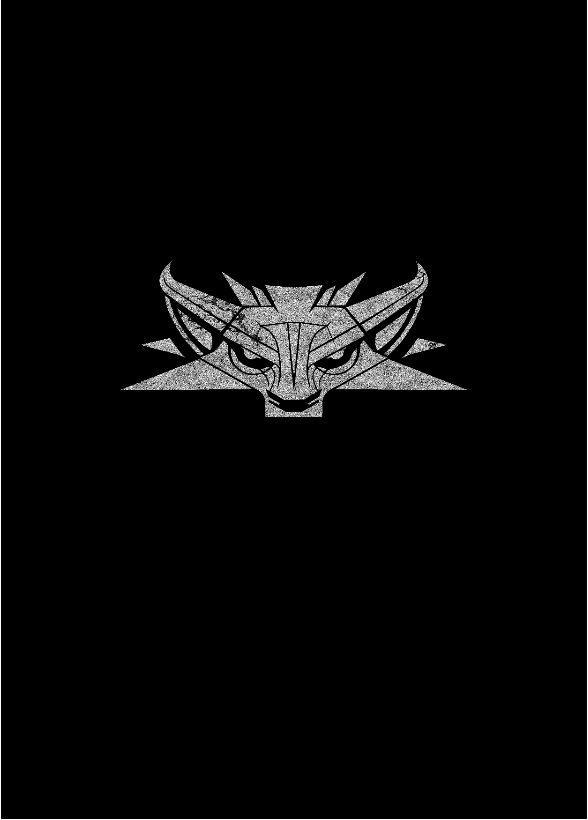
To test the Sequential first I added a clock to the start and end of the code, and this allowed me to time how long it took to perform the task. I also will be making sure that when run the correct images are being output. To add the clock, I first added a startclock to the beginning of the function and I then added a stopclock to the end of the function once the images were saved. To calculate the time taken I then took the endclock time and subtracted the startclock time to give me the time taken in seconds and used cout to display this.

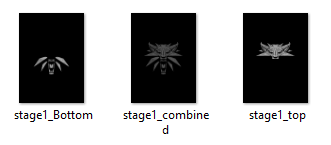






Here you can see that when the sequential code is running, and it saves all 3 of the outputs and has taken 4.758 seconds to perform this task this is as each stage has to me loaded one by one increasing the overall time.

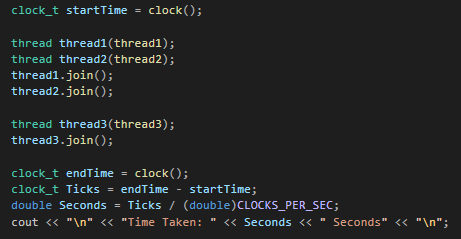
  

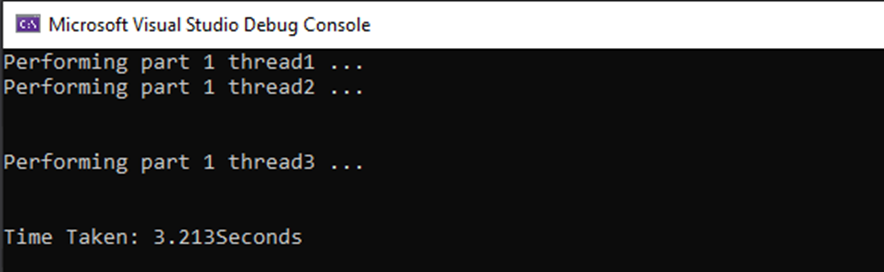


Here you can see the images that were created with the sequential code and the images were all save under the correct names. The image is grey in the combined image as it is taking 50% of the pixels from both images.

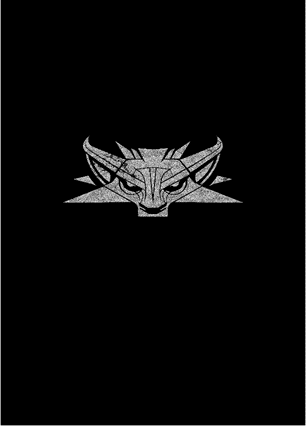
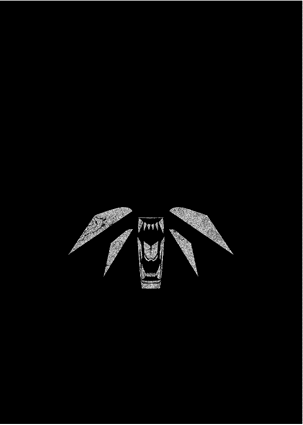
**C++11 multithreading**

To test the C++11 multithreading code I also added a clock to time how long it took to complete the task, and this would give me the ability to compare both versions to see the time difference and conclude the most efficient. I added the clock in the main for this to enable to time for how long it takes to complete thread1 to thread3. Shown below.





Here you can see when I have run the C++11 multithreading code it has loaded both thread 1 and thread 2 and after then run thread 3. The task took 3.213 seconds to complete; this is because the top and the bottom were loaded at the same time there for saving time.





Here you can see the images and file names created when running the C++11 multithreading code as you can see, they are the same as the ones in the sequential version.

**Comparison**

Overall, we can see that both versions of the code produce the correct images and name the files the correct name. The sequential code took 4.758 seconds were as the multi-threaded code only took 3.313 seconds showing that the multithread was slightly fast by 1.545 seconds meaning that the task was preformed amount 30% fast which is quite a significant difference. This was due to sequential version must wait for the top to load before the bottom could start, whereas the multi-threaded version could do both top and bottom at the same time saving time.

Part 2

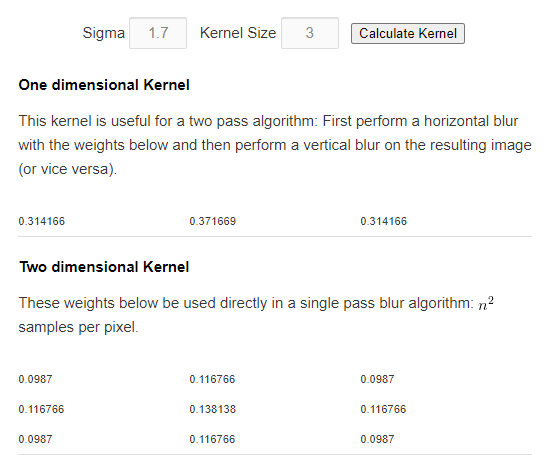
In part 2 I will be applying a blur in 2 ways sequentially and using TBB to parallelise the code. To do this I will be using a stencil pattern and implementing blur imaging using a gaussian kernel. This will then be saved as "stage2\_blurred.png".

I will also be applying a threshold to the blurred image after it has been created also in both ways sequentially and using Tbb. The threshold will make any pixel that is not black to white and will be saved as stage2\_threshold.png".

**Blur sequential**

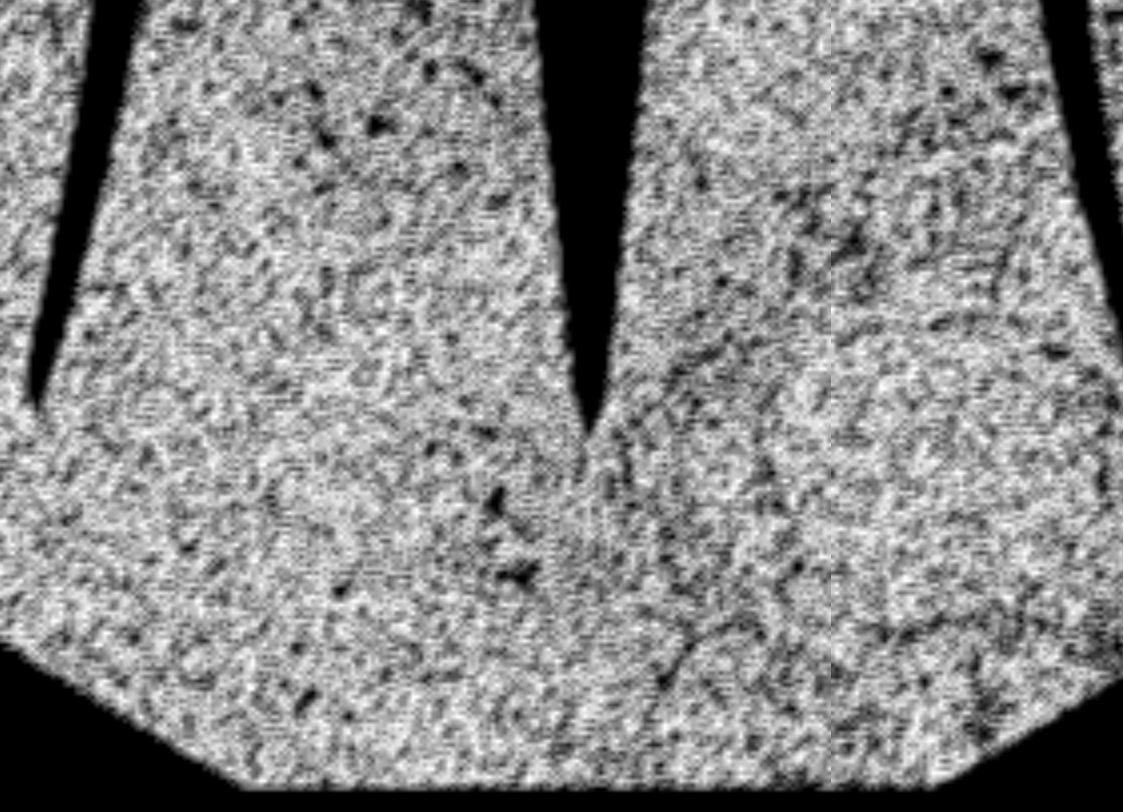


Here you can see the sequential version. I started by firstly loading the combined image created in part 1 in and getting the widths and heights and creating an inputbuffer that accesses the pixels of the image. I then created 2 for loops that look at the width and the height of the image y representing the height and x representing the width making sure that x and y are not more than the height or width. Within the for loop, I set the outputbuffer [y \* width \* x] which allows it to set up a 2d array by supplying a 1d value, to equally the inputbuffer that is split up nine parts times the kernel values that were calculated using kernel calculator shown below.

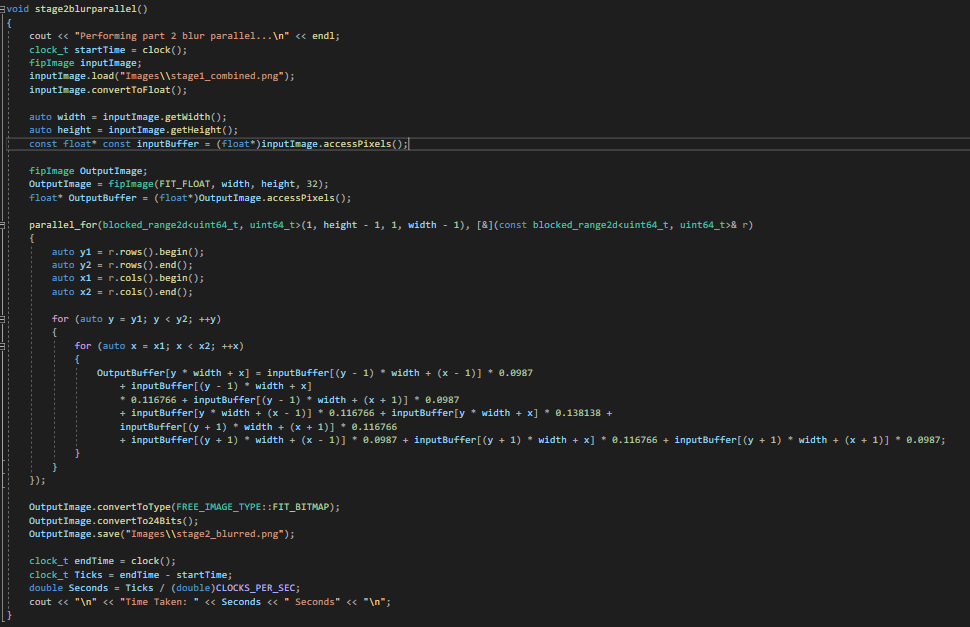


For my blur I used a sigma size of 1.7 and a kernel size of 3 and hard code these values into my blur loop. If you wanted to change the amount of blur, then you could enter different sigma size into the kernel calculator and change the values hard coded.

Once the blur was applied, I saved the image as I have in part one and called it “stage2\_blurred.png”. This is the outputted image with a blur as you can see there is a very soft blur on the image that can be seen when I zoom in on the pixels. See below the blurred image and a zoomed in picture of the blurred pixels.



**Blur parallel**



Here you can see the parallel version. It starts the same as the start of the sequential version where I load in the combined image and create an output image array. I then use Tbb parallel for and used blocked range2D approach which I then supplied the supply it with the rows and the columns of the data from the image and set up a lambda function starting at [&]. I then set the starting and end points for the rows and columns. Within the blocked range each will be sent to them own threads to be processed and will be processing small chunks of data. then the same code is used as the sequential where I set the outputbuffer [y \* width \* x] which allows it to set up a 2d array by supplying a 1d value, to equally the inputbuffer that is split up nine parts times the kernel values.

I then save the image the same as the sequential one and got the exact same output images with the same blur as I am using the same kernel and sigma values.

**Blur testing and comparison**

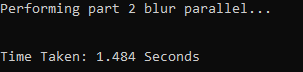
The same as part 1 created a clock to start and stop at the beginning and end of the task to time how long each version of the code has taken to complete this will allow be to test efficiency of the code.

**Sequential**



Here you can see that when running the sequential code, it has taken 1.79 seconds to complete.

**Parallel**



Here you can see that when running the parallel code, it has taken 1.484 seconds to complete as it is running across multiple threads.

**Conclusion**

Overall, we can see that both versions of the code produce the correct images and name the files correctly named. The parallel code is more efficient as it is preforming the task in 1.484 seconds and the sequential is taking 1.79 seconds. although this seems to be a small different of 0.306 seconds the parallel code is just under 20% faster which is a lot faster especially if preforming a large task.

**Threshold sequential**

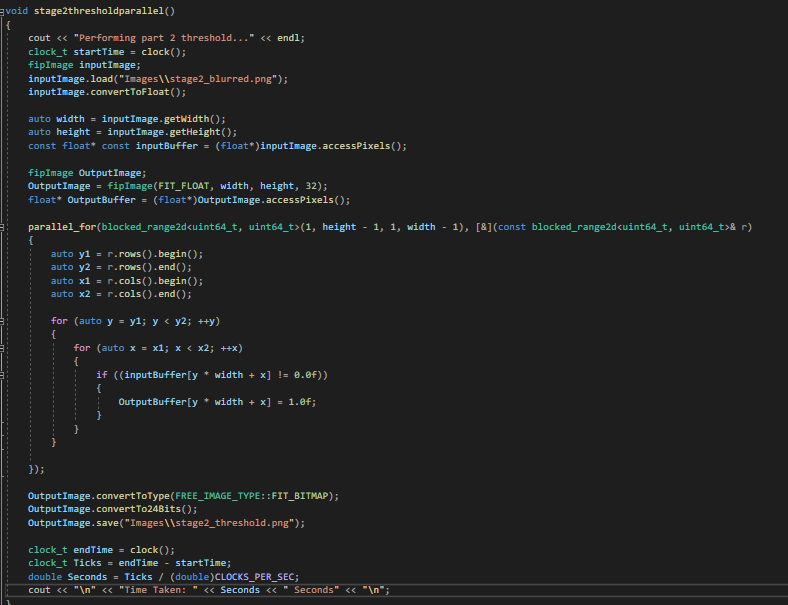


For the sequential threshold I first loaded the blurred imaging using the same load function as shown in part 1 and got the widths and height of the image and times them together to get the number of elements. I then created a for loop that set I to 0 and checked if I was less that numelements and incremented it, this made it check all the pixel in the image and if I was more than numelements then then all the pixels have been checked. The if statement checks to see if the pixel is not black and if its not then it will set it to white.

Then I have saved the image as “stage2\_threshold.png”. Below is the image that Is created that has set all nonblack pixels to white.



**Threshold parallel**



Here is the parallel version of the code like the sequential it uses it loads the blurred image the same but then like the blur parallel it uses a stencil pattern. The Tbb parallel for and used blocked range2D approach which I then supplied the supply it with the rows and the columns of the data from the image and set up a lambda function starting at [&]. I then set the starting and end points for the rows and columns. Within the blocked range each will be sent to them own threads to be processed and will be processing small chunks of data. Then same as the sequential code it will look at the pixels and if they are not black it will set them to white.

Then it saves the images the same as the sequential code and creates the same image but runs faster as the code is more efficient.

**Threshold testing and comparison**

**Sequential**



Here you can see that when running the coding sequentially it took 1.205 seconds to complete the task.

**Parallel**



When running the parallel code, it took 1.187 seconds to complete the task.

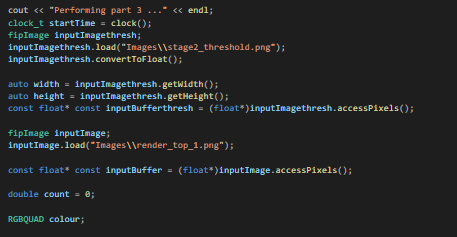
**Conclusion**

Overall, from running both versions of the code we can see that there is not a big difference in time between the parallel and sequential version of the code. With the sequential taking 1.205 seconds and the parallel taking 1.187 second although It has slightly improved the efficiency of the code.

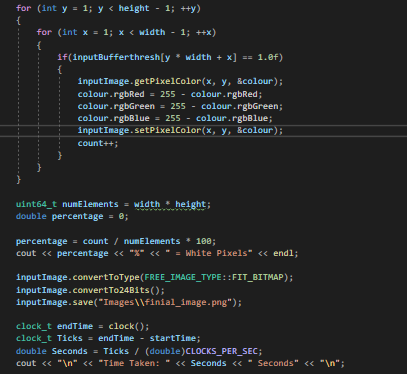
Part 3

In part 3 I will be counting the number of white pixels in stage2\_threshold.png and displaying it as a percentage of white pixels and I will be applying stage2\_threshold.png as a a filter mask invert the pixel colour in render\_top\_1.png at the pixel locations where a white pixel would be on the image "stage2\_threshold.png. I will be doing this in both a sequential and parallel way.

**Sequential**



Here you can see my part 3 sequential I have started by loading the 2 images that are needed stage2\_threshold.png and render\_top\_1.png and I have got the width and heigh so that I can use it later to calculate the percentage of white pixels in stage2\_threshold.png. I have also created a double called count that will be used to store the number of white pixels in the image. I have also used RGBQUAD as this will allow me to get descriptions of consisting of relative intensities of red, green and blue.



I have then created 2 for loop that defines x and y and sets them to 1 and check to make sure that y is not greater than the height and x is not greater than the width and if not, they are incremented. The stencil pattern is used to iterated though the pixels. Then the code will check to see if the pixel in stage2\_threshold.png is equal to 1.0f (white) and if it is it will get the pixel colour rgbRed, rgbGreen and rgbBlue values for render\_top\_1. These RGB values for red, green and blue will be inverted by 255 – the current rgb values for red, green and blue and then sets the new pixel colour on the render\_top\_1.png. The count will then increment for every white pixel to keep a count of the number of white pixels in the image.

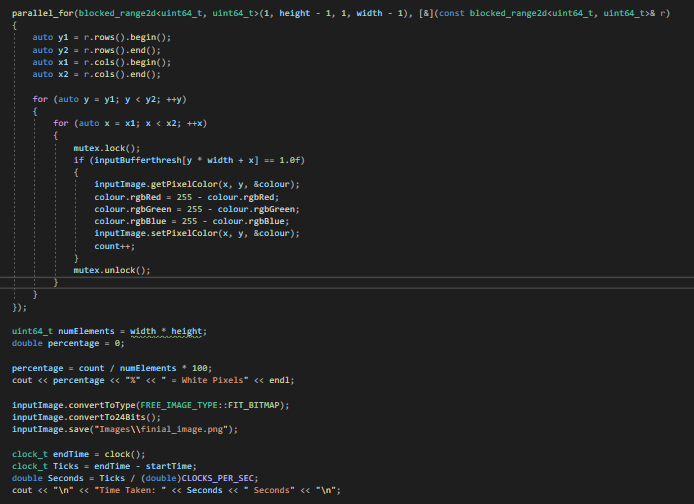
To calculate the number of white pixels in stage2\_threshold.png we have a count of the number of pixels in that image. We then divide the number of white pixels by the number of total pixels to give the percentages of white pixels in the image. And we lastly save the image, and this is the outputted image when the code is run.



**Parallel**



Here you can see the parallel code as you can see that the loading of the images and setting up the count is the same as the sequential version, but I also loaded the mutex function allowing me to lock and unlocking only allowing one thread to access it at one time.

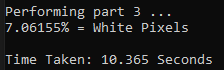


I then set up the Tbb parallel for and used blocked range2D approach which I then supplied the supply it with the rows and the columns of the data from the image. I then set the starting and end points for the rows and columns. Within the blocked range each will be sent to them own threads to be processed and will be processing small chunks of data. I use mutex.lock to lock it so that only one thread can access it and then the code in the if statement is the same as the sequential. I then use mutex.unlock when I have completed the invert filter.

Finally, I then outputted the percentage of white pixels and saved the image. The saved image is the same as the image for the sequential.

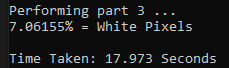
**Testing and conclusion**

**Sequential**



When running the sequential code you can see that there was 7.06115% white pixels in the threshold image and it took 10.365 seconds to calculate the percentage of white pixels and to apply the inverse mask.

**Parallel**



When running the parallel code, you cand see that there were 7.06155% white pixels in the threshold image, and it took 17.973 second to calculate the percentage of white pixels and to apply the inverse mask.

**Conclusion**

Overall, you can see from the result that both version of the code completed the task displaying the correct image and calculating that they are 7.06155% white pixels. But surprisingly the sequential code was faster by over 7 seconds with the sequential completing in 10.365 second and parallel completing it in 17.973 second. Therefor the sequential code is more efficient in this situation.

**Appendix**

#include <iostream>

#include <vector>

//Thread building blocks library

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

//Free Image library

#include <FreeImagePlus.h>

#include <chrono>

#include <tbb/parallel\_for.h>

#include <thread>

#include <mutex>

#define \_USE\_MATH\_DEFINES

#include <tbb/blocked\_range.h>

#include <tbb/blocked\_range2d.h>

#include <cstdlib>

#include <math.h>

#include <random>

#include <cstdint>

#include <functional>

#include <string>

using namespace std;

using namespace tbb;

using namespace std::chrono;

void stage1\_sequential()

{

clock\_t startTime = clock();

cout << "Performing part 1 ..." << endl;

fipImage inputImage;

inputImage.load("Images\\render\_top\_1.png");

inputImage.convertToFloat();

auto width = inputImage.getWidth();

auto height = inputImage.getHeight();

const float\* const inputBuffer = (float\*)inputImage.accessPixels();

fipImage inputImage2;

inputImage2.load("Images\\render\_top\_2.png");

inputImage2.convertToFloat();

const float\* const inputBuffer2 = (float\*)inputImage2.accessPixels();

fipImage inputImage3;

inputImage3.load("Images\\render\_bottom\_1.png");

inputImage3.convertToFloat();

const float\* const inputBuffer3 = (float\*)inputImage3.accessPixels();

fipImage inputImage4;

inputImage4.load("Images\\render\_bottom\_2.png");

inputImage4.convertToFloat();

const float\* const inputBuffer4 = (float\*)inputImage4.accessPixels();

fipImage outputImage;

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

float\* outputBuffer = (float\*)outputImage.accessPixels();

fipImage outputImageTop;

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

float\* outputBufferTop = (float\*)outputImageTop.accessPixels();

fipImage outputImageBottom;

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

float\* outputBufferBottom = (float\*)outputImageBottom.accessPixels();

uint64\_t numElements = width \* height;

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

{

if ((inputBuffer[i] == inputBuffer2[i]))

{

outputBufferTop[i] = 0.0f;

}

else

{

outputBufferTop[i] = 1.0f;

}

if ((inputBuffer3[i] == inputBuffer4[i]))

{

outputBufferBottom[i] = 0.0f;

}

else

{

outputBufferBottom[i] = 1.0f;

}

outputBuffer[i] = outputBufferTop[i] + outputBufferBottom[i];

}

cout << "Part 1 - Saving stage1\_Top, stage1\_Bottom, stage1\_Combined \n";

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

outputImage.convertTo24Bits();

outputImage.save("Images\\stage1\_combined.png");

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

outputImageTop.convertTo24Bits();

outputImageTop.save("Images\\stage1\_top.png");

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

outputImageBottom.convertTo24Bits();

outputImageBottom.save("Images\\stage1\_Bottom.png");

clock\_t endTime = clock();

clock\_t Ticks = endTime - startTime;

double Seconds = Ticks / (double)CLOCKS\_PER\_SEC;

cout << "\n" << "Time Taken: " << Seconds << " Seconds" << "\n";

}

void thread1()

{

cout << "Performing part 1 thread1 ...\n" << endl;

fipImage inputImage;

inputImage.load("Images\\render\_top\_1.png");

inputImage.convertToFloat();

auto width = inputImage.getWidth();

auto height = inputImage.getHeight();

const float\* const inputBuffer = (float\*)inputImage.accessPixels();

fipImage inputImage2;

inputImage2.load("Images\\render\_top\_2.png");

inputImage2.convertToFloat();

const float\* const inputBuffer2 = (float\*)inputImage2.accessPixels();

fipImage outputImageTop;

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

float\* outputBufferTop = (float\*)outputImageTop.accessPixels();

fipImage outputImage;

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

float\* outputBuffer = (float\*)outputImage.accessPixels();

uint64\_t numElements = width \* height;

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

{

if ((inputBuffer[i] == inputBuffer2[i]))

{

outputBufferTop[i] = 0.0f;

}

else

{

outputBufferTop[i] = 1.0f;

}

}

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

outputImageTop.convertTo24Bits();

outputImageTop.save("Images\\stage1\_top.png");

}

void thread2()

{

cout << "Performing part 1 thread2 ...\n" << endl;

fipImage inputImage3;

inputImage3.load("Images\\render\_bottom\_1.png");

inputImage3.convertToFloat();

auto width = inputImage3.getWidth();

auto height = inputImage3.getHeight();

const float\* const inputBuffer = (float\*)inputImage3.accessPixels();

const float\* const inputBuffer3 = (float\*)inputImage3.accessPixels();

fipImage inputImage4;

inputImage4.load("Images\\render\_bottom\_2.png");

inputImage4.convertToFloat();

const float\* const inputBuffer4 = (float\*)inputImage4.accessPixels();

fipImage outputImage;

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

float\* outputBuffer = (float\*)outputImage.accessPixels();

fipImage outputImageBottom;

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

float\* outputBufferBottom = (float\*)outputImageBottom.accessPixels();

uint64\_t numElements = width \* height;

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

{

if ((inputBuffer3[i] == inputBuffer4[i]))

{

outputBufferBottom[i] = 0.0f;

}

else

{

outputBufferBottom[i] = 1.0f;

}

}

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

outputImageBottom.convertTo24Bits();

outputImageBottom.save("Images\\stage1\_Bottom.png");

}

void thread3()

{

cout << "Performing part 1 thread3 ...\n" << endl;

fipImage inputImage;

inputImage.load("Images\\stage1\_Bottom.png");

inputImage.convertToFloat();

auto width = inputImage.getWidth();

auto height = inputImage.getHeight();

const float\* const inputBuffer = (float\*)inputImage.accessPixels();

fipImage inputImage2;

inputImage2.load("Images\\stage1\_top.png");

inputImage2.convertToFloat();

const float\* const inputBuffer2 = (float\*)inputImage2.accessPixels();

fipImage outputImage;

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

float\* outputBuffer = (float\*)outputImage.accessPixels();

uint64\_t numElements = width \* height;

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

{

outputBuffer[i] = inputBuffer[i] + inputBuffer2[i];

}

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

outputImage.convertTo24Bits();

outputImage.save("Images\\stage1\_combined.png");

}

void stage2blurparallel()

{

cout << "Performing part 2 blur parallel...\n" << endl;

clock\_t startTime = clock();

fipImage inputImage;

inputImage.load("Images\\stage1\_combined.png");

inputImage.convertToFloat();

auto width = inputImage.getWidth();

auto height = inputImage.getHeight();

const float\* const inputBuffer = (float\*)inputImage.accessPixels();

fipImage OutputImage;

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

float\* OutputBuffer = (float\*)OutputImage.accessPixels();

parallel\_for(blocked\_range2d<uint64\_t, uint64\_t>(1, height - 1, 1, width - 1), [&](const blocked\_range2d<uint64\_t, uint64\_t>& r)

{

auto y1 = r.rows().begin();

auto y2 = r.rows().end();

auto x1 = r.cols().begin();

auto x2 = r.cols().end();

for (auto y = y1; y < y2; ++y)

{

for (auto x = x1; x < x2; ++x)

{

OutputBuffer[y \* width + x] = inputBuffer[(y - 1) \* width + (x - 1)] \* 0.0987

+ inputBuffer[(y - 1) \* width + x]

\* 0.116766 + inputBuffer[(y - 1) \* width + (x + 1)] \* 0.0987

+ inputBuffer[y \* width + (x - 1)] \* 0.116766 + inputBuffer[y \* width + x] \* 0.138138 +

inputBuffer[(y + 1) \* width + (x + 1)] \* 0.116766

+ inputBuffer[(y + 1) \* width + (x - 1)] \* 0.0987 + inputBuffer[(y + 1) \* width + x] \* 0.116766 + inputBuffer[(y + 1) \* width + (x + 1)] \* 0.0987;

}

}

});

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

OutputImage.convertTo24Bits();

OutputImage.save("Images\\stage2\_blurred.png");

clock\_t endTime = clock();

clock\_t Ticks = endTime - startTime;

double Seconds = Ticks / (double)CLOCKS\_PER\_SEC;

cout << "\n" << "Time Taken: " << Seconds << " Seconds" << "\n";

}

void stage2blursequential()

{

cout << "Performing part 2 blur squential..." << endl;

clock\_t startTime = clock();

fipImage inputImage;

inputImage.load("Images\\stage1\_combined.png");

inputImage.convertToFloat();

auto width = inputImage.getWidth();

auto height = inputImage.getHeight();

const float\* const inputBuffer = (float\*)inputImage.accessPixels();

fipImage OutputImage;

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

float\* OutputBuffer = (float\*)OutputImage.accessPixels();

for (int y = 1; y < height - 1; ++y)

{

for (int x = 1; x < width - 1; ++x)

{

OutputBuffer[y \* width + x] = inputBuffer[(y - 1) \* width + (x - 1)] \* 0.0987

+ inputBuffer[(y - 1) \* width + x]

\* 0.116766 + inputBuffer[(y - 1) \* width + (x + 1)] \* 0.0987

+ inputBuffer[y \* width + (x - 1)] \* 0.116766 + inputBuffer[y \* width + x] \* 0.138138 +

inputBuffer[(y + 1) \* width + (x + 1)] \* 0.116766

+ inputBuffer[(y + 1) \* width + (x - 1)] \* 0.0987 + inputBuffer[(y + 1) \* width + x] \* 0.116766 + inputBuffer[(y + 1) \* width + (x + 1)] \* 0.0987;

}

}

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

OutputImage.convertTo24Bits();

OutputImage.save("Images\\stage2\_blurred.png");

clock\_t endTime = clock();

clock\_t Ticks = endTime - startTime;

double Seconds = Ticks / (double)CLOCKS\_PER\_SEC;

cout << "\n" << "Time Taken: " << Seconds << " Seconds" << "\n";

}

void stage2threshold()

{

cout << "Performing part 2 threshold..." << endl;

clock\_t startTime = clock();

fipImage inputImage;

inputImage.load("Images\\stage2\_blurred.png");

inputImage.convertToFloat();

auto width = inputImage.getWidth();

auto height = inputImage.getHeight();

const float\* const inputBuffer = (float\*)inputImage.accessPixels();

uint64\_t numElements = width \* height;

fipImage OutputImage;

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

float\* OutputBuffer = (float\*)OutputImage.accessPixels();

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

{

if ((inputBuffer[i] != 0.0f))

{

OutputBuffer[i] = 1.0f;

}

}

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

OutputImage.convertTo24Bits();

OutputImage.save("Images\\stage2\_threshold.png");

clock\_t endTime = clock();

clock\_t Ticks = endTime - startTime;

double Seconds = Ticks / (double)CLOCKS\_PER\_SEC;

cout << "\n" << "Time Taken: " << Seconds << " Seconds" << "\n";

}

void stage2thresholdparallel()

{

cout << "Performing part 2 threshold..." << endl;

clock\_t startTime = clock();

fipImage inputImage;

inputImage.load("Images\\stage2\_blurred.png");

inputImage.convertToFloat();

auto width = inputImage.getWidth();

auto height = inputImage.getHeight();

const float\* const inputBuffer = (float\*)inputImage.accessPixels();

fipImage OutputImage;

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

float\* OutputBuffer = (float\*)OutputImage.accessPixels();

parallel\_for(blocked\_range2d<uint64\_t, uint64\_t>(1, height - 1, 1, width - 1), [&](const blocked\_range2d<uint64\_t, uint64\_t>& r)

{

auto y1 = r.rows().begin();

auto y2 = r.rows().end();

auto x1 = r.cols().begin();

auto x2 = r.cols().end();

for (auto y = y1; y < y2; ++y)

{

for (auto x = x1; x < x2; ++x)

{

if ((inputBuffer[y \* width + x] != 0.0f))

{

OutputBuffer[y \* width + x] = 1.0f;

}

}

}

});

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

OutputImage.convertTo24Bits();

OutputImage.save("Images\\stage2\_threshold.png");

clock\_t endTime = clock();

clock\_t Ticks = endTime - startTime;

double Seconds = Ticks / (double)CLOCKS\_PER\_SEC;

cout << "\n" << "Time Taken: " << Seconds << " Seconds" << "\n";

}

void part3sequential()

{

cout << "Performing part 3 ..." << endl;

clock\_t startTime = clock();

fipImage inputImagethresh;

inputImagethresh.load("Images\\stage2\_threshold.png");

inputImagethresh.convertToFloat();

auto width = inputImagethresh.getWidth();

auto height = inputImagethresh.getHeight();

const float\* const inputBufferthresh = (float\*)inputImagethresh.accessPixels();

fipImage inputImage;

inputImage.load("Images\\render\_top\_1.png");

const float\* const inputBuffer = (float\*)inputImage.accessPixels();

double count = 0;

RGBQUAD colour;

for (int y = 1; y < height - 1; ++y)

{

for (int x = 1; x < width - 1; ++x)

{

if(inputBufferthresh[y \* width + x] == 1.0f)

{

inputImage.getPixelColor(x, y, &colour);

colour.rgbRed = 255 - colour.rgbRed;

colour.rgbGreen = 255 - colour.rgbGreen;

colour.rgbBlue = 255 - colour.rgbBlue;

inputImage.setPixelColor(x, y, &colour);

count++;

}

}

}

uint64\_t numElements = width \* height;

double percentage = 0;

percentage = count / numElements \* 100;

cout << percentage << "%" << " = White Pixels" << endl;

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

inputImage.convertTo24Bits();

inputImage.save("Images\\finial\_image.png");

clock\_t endTime = clock();

clock\_t Ticks = endTime - startTime;

double Seconds = Ticks / (double)CLOCKS\_PER\_SEC;

cout << "\n" << "Time Taken: " << Seconds << " Seconds" << "\n";

}

void part3parallel()

{

cout << "Performing part 3 ..." << endl;

clock\_t startTime = clock();

fipImage inputImagethresh;

inputImagethresh.load("Images\\stage2\_threshold.png");

inputImagethresh.convertToFloat();

auto width = inputImagethresh.getWidth();

auto height = inputImagethresh.getHeight();

const float\* const inputBufferthresh = (float\*)inputImagethresh.accessPixels();

fipImage inputImage;

inputImage.load("Images\\render\_top\_1.png");

const float\* const inputBuffer = (float\*)inputImage.accessPixels();

double count = 0;

RGBQUAD colour;

mutex mutex;

parallel\_for(blocked\_range2d<uint64\_t, uint64\_t>(1, height - 1, 1, width - 1), [&](const blocked\_range2d<uint64\_t, uint64\_t>& r)

{

auto y1 = r.rows().begin();

auto y2 = r.rows().end();

auto x1 = r.cols().begin();

auto x2 = r.cols().end();

for (auto y = y1; y < y2; ++y)

{

for (auto x = x1; x < x2; ++x)

{

mutex.lock();

if (inputBufferthresh[y \* width + x] == 1.0f)

{

inputImage.getPixelColor(x, y, &colour);

colour.rgbRed = 255 - colour.rgbRed;

colour.rgbGreen = 255 - colour.rgbGreen;

colour.rgbBlue = 255 - colour.rgbBlue;

inputImage.setPixelColor(x, y, &colour);

count++;

}

mutex.unlock();

}

}

});

uint64\_t numElements = width \* height;

double percentage = 0;

percentage = count / numElements \* 100;

cout << percentage << "%" << " = White Pixels" << endl;

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

inputImage.convertTo24Bits();

inputImage.save("Images\\finial\_image.png");

clock\_t endTime = clock();

clock\_t Ticks = endTime - startTime;

double Seconds = Ticks / (double)CLOCKS\_PER\_SEC;

cout << "\n" << "Time Taken: " << Seconds << " Seconds" << "\n";

}

int main()

{

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

task\_scheduler\_init T(nt);

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

stage1\_sequential();

/\*clock\_t startTime = clock();

thread thread1(thread1);

thread thread2(thread2);

thread1.join();

thread2.join();

thread thread3(thread3);

thread3.join();

clock\_t endTime = clock();

clock\_t Ticks = endTime - startTime;

double Seconds = Ticks / (double)CLOCKS\_PER\_SEC;

cout << "\n" << "Time Taken: " << Seconds << " Seconds" << "\n";\*/

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

//stage2blurparallel();

stage2blursequential();

stage2threshold();

//stage2thresholdparallel();

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

part3sequential();

//part3parallel();

return 0;

}