Multithreaded Image Morphology

Abdul Wasay
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
e-mail: wasay@oakland.edu

***Abstract*—This report presents a program that implements morphological operations, specifically erosion and dilation, using sequential and parallel approaches. Parallelization is achieved using Intel's Threading Building Blocks (TBB) parallel_pipeline construct. The primary aim is to demonstrate improved performance with parallel processing while ensuring correctness through MATLAB-based verification. The report includes detailed descriptions of the implementations, experimental setup, results, and conclusions.**

I. Introduction

Image processing is an important component of modern computational systems. As image resolutions and data sizes continue to grow, the computational requirements for processing these images also increase, making speed and efficiency a significant concern. Many image processing operations, such as morphological transformations, are inherently parallelizable, as they involve repetitive calculations over large data arrays. By introducing parallel processing frameworks we can significantly reduce execution times for these image processing operations.

This project focuses on implementing a suite of image processing operations, including erosion, dilation, boundary extraction, opening, and closing to evaluate the potential speedup achieved through parallelization. Both sequential and parallel implementations of these operations were developed, with the parallel implementations utilizing Intel's Threading Building Blocks (TBB) library. The TBB structure offers high-level functions like parallel_pipeline and parallel_for, which are well-suited for tasks that can be parallelized and divided into multiple independent stages, such as row-by-row image processing.

The primary goal of this project is to analyze and compare the performance of sequential and parallel implementations for each morphology operation. Metrics such as execution time and time difference between both implementations are used to evaluate the effectiveness of the parallelization strategy. By demonstrating the benefits of parallel processing, this program highlights the importance of optimizing computational speed and efficiency when performing image processing.

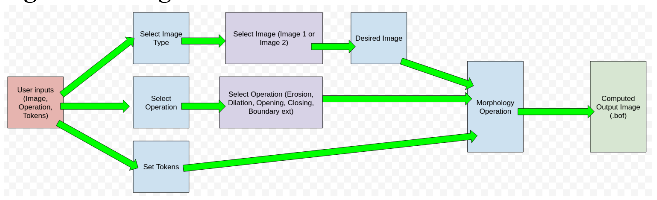
II. Methodology

This section elaborates on the program design, the methods used for parallelization, and the detailed implementation of each image processing operation. It describes the program's structure, inputs and outputs, and the steps for each operation, both sequentially and in parallel.

A. Main File

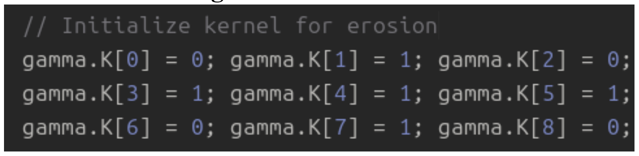
The main file is the heart of the program. It handles the user's command-line inputs for selecting the programs inputs: the number of tokens to use with parallel_pipeline, the image to perform the operations on, the type of operation to perform (erosion, dilation, opening, closing, boundary extraction). Based on user input, it invokes the corresponding functions from one of the program's header files, finalproject_fun.cpp. Additionally, the program will perform both the sequential operation along with the parallelized one. It then compares the processing times from sequential and parallel implementations of the two processing methods. The main file will then print the completed operating times between the methods and compute the difference to show the user the performance gains of using TBB parallel_pipeline vs sequential operation. FIGURE 1 shows the program flow. First the user enters in all of the program inputs (ntokens, image type, operation). The program will then select the image and set the parameters accordingly. In addition it will also select which operation to use and set the image kernel based on the operation. Once that step is done, the program will pass all the information to the morphology function and perform the calculation. Finally, it will output the morphed image in a binary output file.

Figure 1 - Program Flow


User inputs (Image, Operation, Tokens)
Select Image Type
Select Image (Image 1 or Image 2)
Desired Image
Select Operation
Select Operation (Erosion, Dilation, Opening, Closing, Boundary ext)
Set Tokens
Morphology Operation
Computed Output Image (.bof)

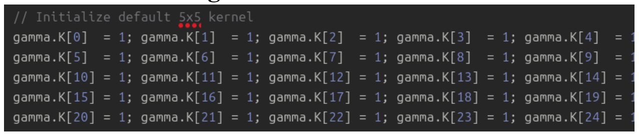
When the program will be invoked, the user will input the number of tokens, desired image type, and desired morphology operation. The main file will then use the selected image to configure the structure elements' parameters. If the user selects image 1, being the smaller 640x480 image, the structure's (X,Y) values along with the kernel size will be updated. Then another check is performed, if the user selects the Erosion operation on image 1, the program will use a smaller 3x3 Kernel as opposed to the 5x5 kernel used for the other operations. FIGURE 2 shows the 3x3 kernel used for image 1's erosion operation. For all other operations the 5x5 kernel shown in FIGURE 3 is used.

Figure 2 - 3x3 kernel



```
// Initialize kernel for erosion
gamma.K[0] = 0; gamma.K[1] = 1; gamma.K[2] = 0;
gamma.K[3] = 1; gamma.K[4] = 1; gamma.K[5] = 1;
gamma.K[6] = 0; gamma.K[7] = 1; gamma.K[8] = 0;
```

Figure 3 - 5x5 kernel

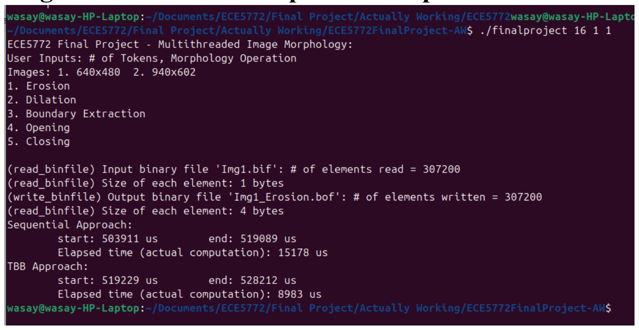


```
// Initialize default 5x5 kernel
gamma.K[0]  = 1; gamma.K[1]  = 1; gamma.K[2]  = 1; gamma.K[3]  = 1; gamma.K[4]  =
gamma.K[5]  = 1; gamma.K[6]  = 1; gamma.K[7]  = 1; gamma.K[8]  = 1; gamma.K[9]  =
gamma.K[10] = 1; gamma.K[11] = 1; gamma.K[12] = 1; gamma.K[13] = 1; gamma.K[14] =
gamma.K[15] = 1; gamma.K[16] = 1; gamma.K[17] = 1; gamma.K[18] = 1; gamma.K[19] =
gamma.K[20] = 1; gamma.K[21] = 1; gamma.K[22] = 1; gamma.K[23] = 1; gamma.K[24] =
```

Once the image parameters are loaded based on the respective image, the program will select the desired morphology operation based on user input. It will first start the timer and then perform the operation sequentially. Once the operation is done the timer will stop and the computation time is stored. After this the output is cleared and the timer will start for the parallel operation. Once this operation is complete, the timer will end. Next the program will name the output file based on the image type and operation. For example, image 1 with erosion performed on it will cause the output file to be named "Img1_Erosion.bof". This is to allow for user ease in keeping track with the output files across the different images and operations.

After the desired computations are completed, the program will output the computation times for both the sequential and parallel_pipeline implementations into the terminal for the user to view. FIGURE 4 shows the terminal output with computation times.

Figure 4 - Terminal Output w/ Computation Times



```
wasay@wasay-HP-Laptop:~/Documents/ECE5772/Final Project/Actually Working/ECE5772wasay@wasay-HP-Lapto
~/Documents/ECE5772/Final Project/Actually Working/ECE5772FinalProject-AW$ ./finalproject 16 1 1
ECE5772 Final Project - Multithreaded Image Morphology:
User Inputs: # of Tokens, Morphology Operation
Images: 1. 640x480  2. 940x602
1. Erosion
2. Dilation
3. Boundary Extraction
4. Opening
5. Closing

(read_binfile) Input binary file 'Img1.bif': # of elements read = 307200
(read_binfile) Size of each element: 1 bytes
(write_binfile) Output binary file 'Img1_Erosion.bof': # of elements written = 307200
(read_binfile) Size of each element: 4 bytes
Sequential Approach:
        start: 503911 us        end: 519089 us
        Elapsed time (actual computation): 15178 us
TBB Approach:
        start: 519229 us        end: 528212 us
        Elapsed time (actual computation): 8983 us
wasay@wasay-HP-Laptop:~/Documents/ECE5772/Final Project/Actually Working/ECE5772FinalProject-AW$
```

B. Function File

This file contains the core implementations for all operations. Each morphological operation: erosion, dilation, opening, closing, and boundary extraction, is implemented with two methods: sequentially and parallelized. The parallel versions utilize Intel TBB parallel_for and parallel_pipeline to divide the workload among multiple threads efficiently. In addition to the morphological operations, this header file also includes the read_binfile() and write_binfile() functions.

These two functions are provided by the instructor and are kept unmodified in this project. The read_binfile() function is used to read incoming image data from the folder structure. The function has 4 inputs. The first is called 'data'. This is where the read image data will be stored. The second function is 'length' which is a measure of the images x*y size. After that we have 'in_file' which is the input file where the image data is stored. Finally the type which is used to define if the input image data is either an 8 bit unsigned integer or a 32 bit signed integer. This function will sequentially index through the image data and store them in the desired image location. write_binfile() is also very similar to read_binfile() The only major difference is that instead of having one input be the input image data file, this has an output file location as an input. The function will write the morphed image data to this location.

The first morphological image operation is Erosion. The erosion operation usually uses a kernel for reducing the shapes contained in the input image. It works by employing a kernel that defines the neighborhood of a pixel. For each pixel, erosion evaluates whether the kernel fits entirely within the object; if not, the pixel is removed. The operation is highly effective for removing small noise, breaking thin connections between objects, and simplifying object shapes. It is performed both sequentially and with parallel_pipeline.

The im_erosion() function is a sequential implementation of the morphological erosion operation, applied to an image represented as a 2D array. This function takes a structure containing the input image (I), the output image (O), the kernel (K), the dimensions of the image (sX and sY), and the dimensions of the kernel (kX and kY). The goal is to compute the eroded image and store it in the output array O.

The main operation is in a nested loop structure. The outer loops iterate over every pixel of the input image (I), identified by row i and column j. For each pixel, the variable min_val is initialized to 255, the maximum possible pixel intensity, so that any valid pixel from the surroundings will replace it during the erosion process. The nested loops iterate over the kernel (K), moving through each of its elements (m and n). The coordinates of the current pixel in the input image are computed as y = i + m - kY / 2 and x = j + n - kX / 2, centering the kernel around the pixel being processed. A boundary check ensures that the kernel does not exceed the image dimensions. If the kernel's value at position (m, n) is 1 and the corresponding image pixel exists, its intensity is compared with min_val. If the pixel's value is smaller, min_val is updated. After examining all pixels within the kernel's neighborhood, the smallest intensity found (min_val) is assigned to the output pixel O[i * sX + j]. This process is repeated for every pixel in the input image, applying the erosion operation.

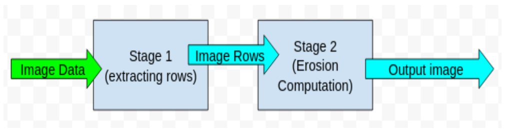
The im_erosion_tbb() is the parallelized implementation of erosion. The function begins by extracting the required data from the structure, including the input image (I), output image (O), structuring element (K), image dimensions (sX, sY), kernel dimensions (kX, kY), and the number of tokens (ntoken). The ntoken parameter determines the number of tokens used and defaults to 16 if not specified.

The first stage of the parallel_pipeline is defined using a *serial filter* (filter_mode::serial_in_order). This stage is responsible for creating tokens that represent rows of the image to be processed. A variable 'row' is used to track the current row, and the stage sequentially increments this variable to produce tokens. If all rows have been processed, the flow is stopped using fc.stop(). This stage ensures that rows are distributed to subsequent pipeline stages in an organized way.

The second stage of the pipeline is defined as a *parallel filter* (filter_mode::parallel), which allows multiple threads to process different rows concurrently. Each thread receives a row index (i) and performs the erosion operation for all pixels in that row. For each pixel (i, j) in the row, the minimum intensity value in the neighborhood defined by the kernel is computed. This involves iterating over the kernel (m and n) and mapping the kernel elements to its corresponding image pixels. Boundary checks make sure that the kernel does not cross the image dimensions, and only valid pixels are compared to update the min_val variable. After all kernel positions have been evaluated, the smallest value is written to the output array O[] at the corresponding location.

Compared to the sequential im_erosion() function, the TBB implementation achieves significant performance improvements by distributing the workload to multiple threads. This parallel processing reduces execution time, especially for large images. The pipeline structure also simplifies the code organization, separating input handling from computation, making the parallelization strategy more modular and easier to understand. FIGURE 5 shows a depiction of the parallel_pipeline structure.

Figure 5 - Erosion Parallel_Pipeline



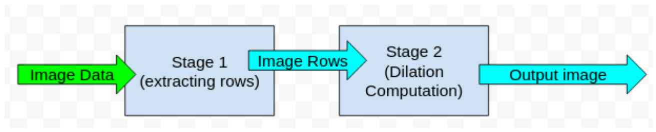
The next function is im_dilate() which is the sequentially implemented dilation operation This function processes an input image pixel by pixel, using a kernel to identify and write the maximum value in the neighborhood defined by the kernel to the output image. The function begins by validating the input parameters. It checks whether the pointers for the input image (I), output image (O), and kernel (K) are not null, ensuring that the required data is available. The dimensions of the input image (sX, sY) and kernel (kX, kY) are validated to ensure they are positive values. It then moves onto the computation portion.

The function loops through each pixel of the input image using two nested loops. The outer loop goes over the rows (i) of the image. The inner loop goes over the columns (j) of the image. For each pixel (i, j), the function determines the maximum value in the neighborhood defined by the kernel. For each pixel, the kernel is applied using two additional nested loops. The kernel rows (m) and columns (n) are looped over. The position of the kernel's elements is calculated using the index y = i + m - kY / 2 and x = j + n - kX / 2. This centers the kernel on the pixel. The value of the pixel in the input image is investigated and the max value in the kernel is written to the output array O[].

The TBB application, im_dilate_tbb() is the parallelized version of this. The parallel_pipeline operation is structured into two stages: row generation and the dilation computation. The number of tokens, defined by ntoken, controls the degree of parallelism. The row generation stage is used to generate rows of the image to be processed in the computational stage. It divides the rows and gives them one row at a time into the pipeline. A variable row keeps track of the current row index. This stage is serialized to make sure the row order is kept. The dilation computation stage applies the dilation operation on each row in parallel. For each pixel (j) in the row (i), the function initializes to the smallest possible integer value. The kernel is looped over, and the input pixel values are used in the calculation. The resulting value is assigned to the pixel's position in the output image. Multiple rows are processed in parallel by different threads, as defined by ntoken. Each thread operates independently on its assigned row maximising efficiency.

After all rows are processed, the pipeline completes, and the function returns 1, indicating successful execution. This parallel implementation is beneficial for large images or kernels, where sequential processing would be much slower. FIGURE 6 shows the parallelized implementation with stages.

Figure 6 - Dilation Parallel_Pipeline



The im_boundary function performs boundary extraction,which is used to highlight the edges of objects in an image. The operation calculates the boundary of an image by subtracting the result of an erosion operation from the original image. This function is implemented sequentially. It begins by checking the validity of the input parameters. It then checks that the pointers for the input image, output image, and kernel are not null. The checks that the dimensions of the input image (sX, sY) and the kernel (kX, kY) are positive.

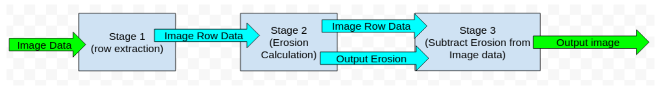
This function also utilizes a temporary buffer. The purpose of which is to store the result of the erosion operation. The buffer is dynamically allocated using calloc to ensure it is initialized to zero. The erosion operation is applied to the input image using the given kernel for each pixel. The function sets min_val to 255, representing the highest intensity for an 8-bit grayscale image. The function calculates the coordinates of the corresponding input image pixel relative to the kernel center. If the kernel is within the image's bounds and the minimum value is updated and stored in the temporary buffer labeled 'eroded'. This process creates an eroded version of the image, which serves as the input for the boundary extraction step. After erosion, the function calculated the boundary by subtracting the eroded image from the original image. This operation isolates the edges of objects by removing the inner portion of each object, leaving only the boundary. The output image is then stored in the output structure array O[];

The im_boundary_tbb function implements boundary extraction with TBB. This implementation performs the erosion operation and boundary calculation in three distinct stages, using parallel_pipeline and parallel_for to maximize efficiency. A temporary buffer (eroded) is allocated using calloc to store the result of the erosion operation. This buffer is used to calculate the boundary in the final stage. The first stage ensures rows are processed sequentially to make sure that the rows are fed in order so that they are not read out of order. A variable, row, tracks the current row being processed. This stage feeds the other rows, passing row indices to the next stage for erosion processing.

The next stage is the erosion computation. In this parallel stage, TBB divides the workload among multiple threads. Each row index from Stage 1 is processed independently, allowing for parallelized calculations. For every pixel in the row. min_val is initialized to 255, the maximum value in an 8-bit grayscale image. It then Iterates over the kernel dimensions to compute the erosion. Once it is made sure that the kernel is within image bounds, the min_val is Updated with the minimum value between min_val and the pixel intensity at pixel's coordinates. The result is then stored in the 'eroded' buffer. This stage being parallelized, significantly speeds up the erosion step. Due to each row and pixel being processed independently.

After erosion, The third stage of the function computes the boundary by subtracting the eroded image from the original image. This is done using tbb parallel_for. The image is divided into row segments for parallel processing. This stage isolates object boundaries by removing the inner portions of objects, leaving only the edges. It is then written to the output array O[] once all indexes of the image have been subtracted. FIGURE 7 shows the flow diagram for im_boundary_tbb().

Figure 7 - Boundary Extraction Parallel_Pipeline



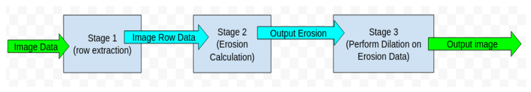
The next function is im_open. This function performs morphological opening on an image. Opening is a combination of erosion followed by dilation, using the same kernel. This function achieves the task in different steps, using temporary storage to do the two morphological operations. The erosion step is executed using the im_erosion function. The image data is processed with the kernel, the result is then stored in the temporary buffer. Erosion reduces object size by discarding boundary pixels that do not completely fit the kernel. The dilation step is executed using the im_dilate function. The intermediate result from tempBuffer is expanded using the same kernel. The result is stored in the output buffer O[]. Dilation after erosion smooths object boundaries and reconnects broken structures, effectively "opening" the image by removing smaller objects while maintaining the larger shapes.

The im_open_tbb function implements opening using TBB to parallelize the operations. Morphological opening is a sequence of erosion followed by dilation, both of which are performed in distinct stages using TBB's parallel_pipeline mechanism. The function begins by validating the input parameters. It checks that the input image, output buffer, kernel, and that image and kernel dimensions are valid. Once the inputs are validated, a temporary buffer named eroded is used to store the intermediate erosion results. This buffer has the same size as the input image and will hold the output of the erosion step, serving as the input for the subsequent dilation.

The erosion step is performed using a two-stage TBB parallel pipeline. The first stage iterates through the image rows sequentially. It ensures that rows are processed in order to be used in the parallel stage. The second stage processes each row in parallel, computing the erosion for all columns within that row. For each pixel, it scans the neighborhood defined by the kernel to find the minimum value among pixels overlapping with the kernel. This value is stored in the eroded buffer.

After completing the erosion, the dilation step is performed in a similar manner using another TBB parallel pipeline. The first stage, sequentially passes rows to the parallel computation stage. The second stage calculates the dilation for each pixel in parallel, using the intermediate eroded buffer as input. For each pixel, it determines the maximum value within the neighborhood defined by the kernel. The result is stored in the output buffer (O). FIGURE 8 shows the flow diagram for im_open_tbb().

Figure 8 - Opening Parallel_Pipeline



The im_close function implements morphological closing sequentially. Morphological closing is an operation consisting of dilation followed by erosion, designed to fill small holes and smooth the contours of objects in an image. This function uses two of the previously defined morphological operations im_dilate and im_erosion to perform this task, and a temporary buffer for intermediate results. The function starts by dynamically allocating a temporary buffer, with a size equal to the number of pixels in the input image. This buffer will store the intermediate results of the dilation step, to make sure that the input image remains unmodified.

The dilation step is performed first. The function uses a structure, which contains parameters for the dilation operation. The output of this structure will save in tempBuffer. The im_dilate function is then called to execute the dilation operation. During this step, each pixel in the image is replaced by the maximum value in its neighborhood defined by the kernel.

After the dilation, the erosion step is executed. A second structure is used, where the input is taken from the temporary buffer, making the dilation result the input for the erosion process. The output is set to the final output buffer, so the results of the erosion step are written to the provided output array. The im_erosion function is then called, and for each pixel, the minimum value in the kernel neighborhood is computed and stored in the output buffer. This completes the closing operation.

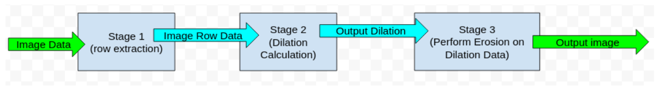
The im_close_tbb function implements closing using a parallelized. Morphological closing involves a dilation operation followed by an erosion operation, with the goal of smoothing object boundaries and filling small holes. This method uses two temporary buffers: one for storing the intermediate dilation result and another for the final erosion result. The function employs TBB parallel_pipeline and parallel_for to parallelize the operations.

Memory is allocated for the first temporary buffer, which will hold the result of the dilation operation. The dilation operation is executed as the first stage using parallel pipeline. The pipeline begins with a serial row selection process, ensuring that rows are distributed to parallel threads in the correct order. The second stage of the pipeline computes the dilation for each selected row in parallel. For every pixel, the function calculates the maximum value within the kernel's neighborhood, constrained by the kernel, and stores the result in the dilated buffer.

Once the dilation is complete, memory is allocated for the second temporary buffer, which will hold the final result of the erosion operation. The erosion step is then performed using another parallel pipeline. Similar to dilation, this pipeline starts with a serial row selection stage, followed by a parallel stage that computes the erosion for each row. For every pixel, the function determines the minimum value within the kernel's neighborhood, storing the result in the eroded buffer. This step uses the dilated buffer as input, thereby completing the closing operation.

Finally, the contents of the eroded buffer are copied to the output image using a parallel_for loop, using parallelism in this step as well. By using TBB's parallelization capabilities, the im_close_tbb function efficiently processes images, making it suitable for high-performance morphological operations on large datasets. FIGURE 9 shows a flow diagram of the parallelized implementation of im_close_tbb.

Figure 9 - Closing Parallel_Pipeline



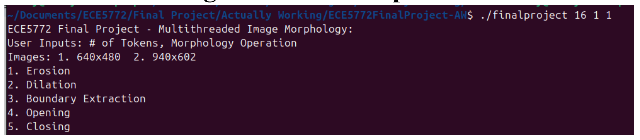
C. *Header File*

This header file defines the structure and function prototypes for image processing operations, supporting both sequential and parallel implementations using Intel's TBB library. It includes the struct, which holds parameters such as image dimensions, kernel dimensions, a token count for parallel pipeline, and pointers to the input image, kernel, and output image. The file provides prototypes for file I/O functions to handle binary image data efficiently. It also declares sequential and TBB-parallelized versions of key image processing functions: erosion, dilation, morphological opening , boundary extraction, and morphological closing. By having all the function prototypes in a single location, the program is much more organized and easier to follow.

III. Experimental Setup

The experimental setup involves compiling and running the C++ image processing program and validating its outputs using MATLAB for morphological operations. This ensures correctness and provides a reference for comparison. The program also outputs the execution times for operations in the terminal, so that the user can perform performance analysis. FIGURE 10 shows the user input into the command line to perform a morphological operation.

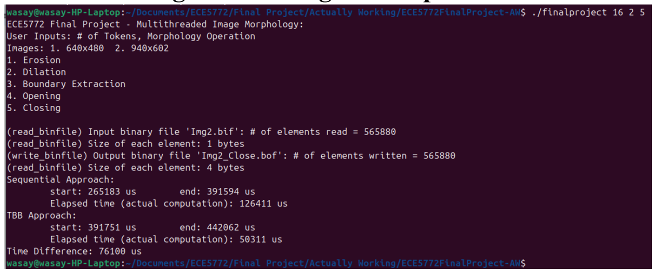
Figure 10 - User input



```
~/Documents/ECE5772/Final Project/Actually Working/ECE5772FinalProject-AW$ ./finalproject 16 1 1
ECE5772 Final Project - Multithreaded Image Morphology:
User Inputs: # of Tokens, Morphology Operation
Images: 1. 640x480  2. 940x602
1. Erosion
2. Dilation
3. Boundary Extraction
4. Opening
5. Closing
```

When the program is compiled, it will print instructions on how to use the tool into the terminal window. As shown in FIGURE 10, it shows all the valid user inputs and what order to enter them in. When an operation is complete, the program prints the operation times for sequential and parallel implementations directly to the terminal. This allows for an easy comparison of performance gains between sequential and parallel versions. FIGURE 11 shows the output of the program when an operation has been completed. It shows the time for the sequential and parallel operations and shows the time difference between the two.

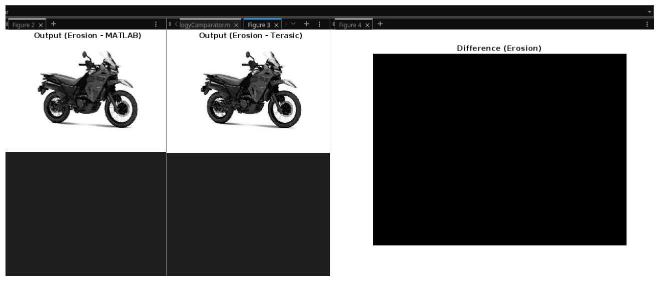
Figure 11 - Program output

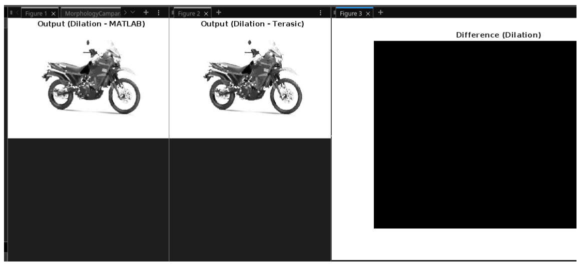


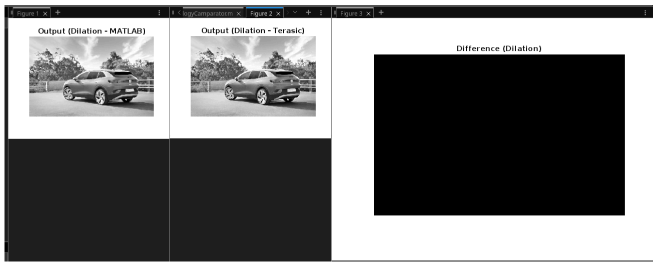
```
wasay@wasay-HP-Laptop:~/Documents/ECE5772/Final Project/Actually Working/ECE5772FinalProject-AW$ ./finalproject 16 2 5
ECE5772 Final Project - Multithreaded Image Morphology:
User Inputs: # of Tokens, Morphology Operation
Images: 1. 640x480  2. 940x602
1. Erosion
2. Dilation
3. Boundary Extraction
4. Opening
5. Closing

(read_binfile) Input binary file 'Img2.bif': # of elements read = 565880
(read_binfile) Size of each element: 1 bytes
(write_binfile) Output binary file 'Img2_Close.bof': # of elements written = 565880
(read_binfile) Size of each element: 4 bytes
Sequential Approach:
        start: 265183 us        end: 391594 us
        Elapsed time (actual computation): 126411 us
TBB Approach:
        start: 391751 us        end: 442062 us
        Elapsed time (actual computation): 50311 us
Time Difference: 76100 us
wasay@wasay-HP-Laptop:~/Documents/ECE5772/Final Project/Actually Working/ECE5772FinalProject-AW$
```

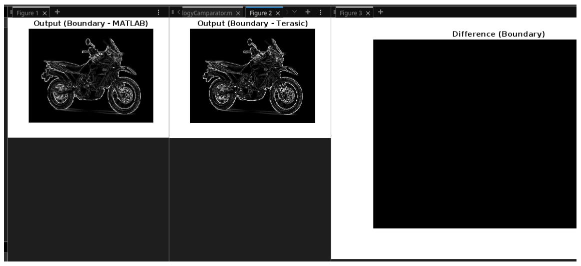
MATLAB is used to validate the c++ programs operation. First the user selects the desired image to compare the image morphology operation on. The selected jpeg file is converted to a .bif, or binary image file, which is read as a 1d array. The matlab script has all the morphological operations found in the c++ program. Such as erosion, dilation, boundary extraction, opening, and closing. The matlab will perform its own morphological operation and then compare it to the one generated by the c++ program. It does so by subtracting the difference between the matlab generated morphed image and the one from the c++. By doing so we are able to see if there are any differences between the matlab and c++ operations. It will display the input image, the matlab generated morphed image, the c++ morphed image, and finally the difference image where the two operations are subtracted. Figure 12 shows the different output images from the matlab.

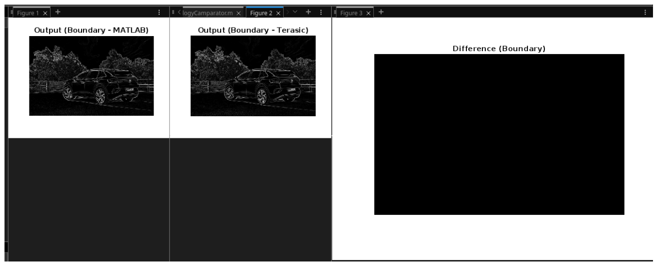
Figure 12 - Morphological Operations

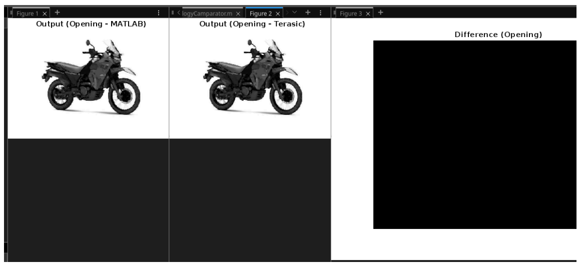


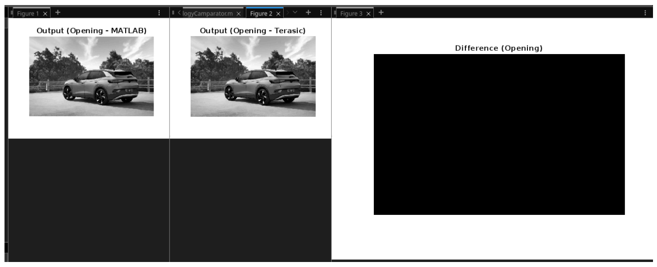


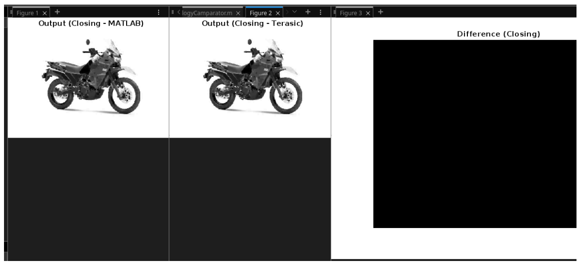


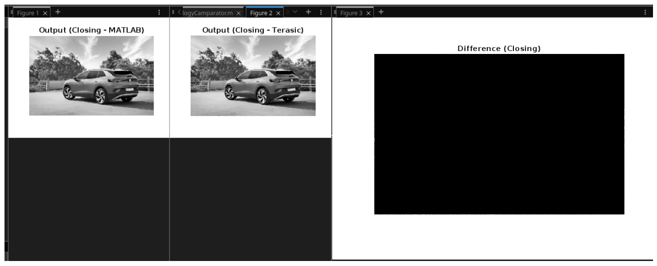


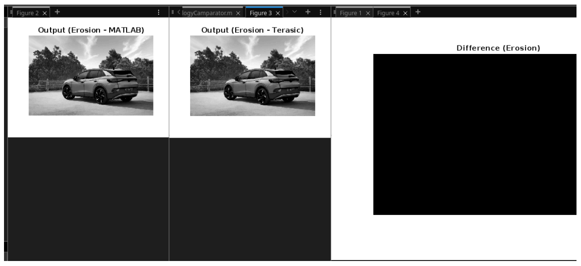












IV. Results

The program performed all of the morphological operations with no issues. In all cases the parallelized approach was much faster than the serial implementation. This however changed in relation to the user input *ntoken*. If the user assigned value was either too small or too large the parallelized operation's processing time would be negatively affected. FIGURE 13 shows a table of the morphological operations computation times with different *ntoken* values (5, 16, 100). When comparing the program and matlab output images, they match for all image and morphology operation cases. As you can see in the table, the parallelized computation times are significantly faster than the sequential operations. In most cases being 2-4 times faster.

In addition we can see that the computation times lower from 5 tokens to 16, but increase, albeit slightly, from 16 tokens to 100.

Figure 13 - Table with Computation Times

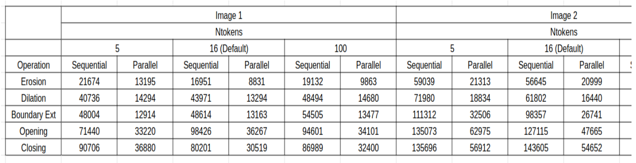


	Image 1						Image 2			
	Ntokens						Ntokens			
	5		16 (Default)		100		5		16 (Default)	
Operation	Sequential	Parallel	Sequential	Parallel	Sequential	Parallel	Sequential	Parallel	Sequential	Parallel
Erosion	21674	13195	16951	8831	19132	9863	59039	21313	56645	20999
Dilation	40736	14294	43971	13294	48494	14680	71980	18834	61802	16440
Boundary Ext	48004	12914	48614	13163	54505	13477	111312	32506	98357	26741
Opening	71440	33220	98426	36267	94601	34101	135073	62975	127115	47665
Closing	90706	36880	80201	30519	86989	32400	135696	56912	143605	54652

Conclusion

This project successfully implemented and validated multithreaded image processing operations using C++ and the Intel TBB library. Morphological operations such as erosion, dilation, boundary extraction, opening, and closing were optimized for parallel execution, showing significant performance improvements over their sequential versions. By including the execution time measurements we are able to accurately measure how big the performance gains actually are.

References

[1] M. Voss, R. Asenjo Plaza, and J. Reinders, *Pro TBB: C++ Parallel Programming with Threading Building Blocks*. New York, NY: APress Open, 2019.

[2] Intel, “Intel® threading building blocks tutorial,” Intel® Threading Building Blocks, https://moodle.oakland.edu/pluginfile.php/9501752/mod_resource/content/1/IntelTBB_tutorial.pdf (accessed 2024).

[3] D. llamocca, “TBB: parallel_pipeline,” Tutorial: High-Performance Embedded Programming with the Intel® AtomTM Platform, https://moodle.oakland.edu/pluginfile.php/9501755/mod_resource/content/3/Tutorial%20-%20Unit%207.pdf (accessed 2024).