Multithreaded Image Processing Tool

ECE 5772 – High Performance Embedded Programming

List of Authors (Mazen Albarazi, Martin Shoraji)
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
e-mails: albarazi@oakland.edu, martinshoraji@oakland.edu

Abstract— **A parallel implementation of image processing operations utilizing the Threading Building Blocks (TBB) library created and compared with a sequential implementation to express the benefits of multithreaded parallel computing. The implementation is done on the Terasic DE2i-150 Development Board. The project includes several image processing operations such as average blurring, Gaussian blurring, edge detection, erosion, dilation, and gamma correction. These operations use concepts such as convolution and multiple other image processing complex techniques and manipulations. The project focuses on comparing thoroughly the computation times of a sequential implementation and the TBB implantation and explores how the parallel implementation benefits and improves the efficiency and reduces computation time for large data sets and on more efficient processors. The code is implemented using different types of data structures in C++ and utilizes libraries as needed to implement the parallelization and extract the computation times. Major findings concluded that parallel programming is a much more efficient approach than sequential for image processing. Different factors such as the image size or number of elements, different processors' core count and speed play a significant role in the operation calculation time.**

I. INTRODUCTION

Image processing is the use of a computer to process images through an algorithm. Since images are defined in two dimensions, digital image processing may be modeled in the form of multidimensional systems such as matrices. The generation and development of the image processing is mainly affected by three factors, the device it is run on, the development of the algorithm, and the application desired. The Objective was to create an image processing tool that the user can use for many of the popular filtering and morphological image operations and compare the processing times for the different operations in respect to a sequential and a parallel approach, as well as to different image sizes and running it on different machines, thus, demonstrating the efficiency of a multithreaded image processing tool over a sequential one.

II. METHODOLOGY

A. Algorithm

Some of the image processing operations used are blur, which includes three types: 3x3 average blur, 5x5 blur, and Gaussian blur. Another operation utilized is edge detection. These operations all use specialized kernels that implement filtering through the convolution concept. Other operations utilized are dilation and erosion, which are morphological operations that also use the convolution concept but also change the shape of the image by adding to or subtracting pixels from the original image. Finally, a gamma correction function is also implemented as part of the project. This operation is used to change the luminance values for each of the image pixels.

Below are some examples of the image operations previously implemented and are explored as part of the project.

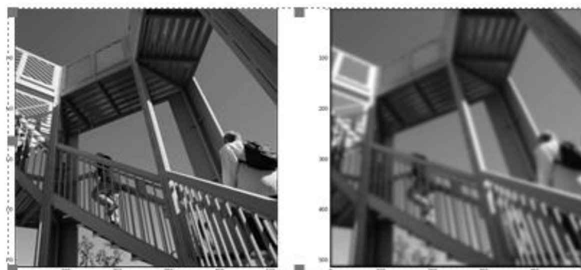

Figure 1. Gaussian Blur

Three of the operations implemented are focused on a blur filter. Figure 1 shows one of the three blur filters, the Gaussian Blur [1], which is a special type of blurring that uses a Gaussian function for calculating the transformation to apply to each pixel in the image.

Figure 2. Edge Detection

Figure 2 shows the edges of an image based on an edge detection filter [2], which is used to compare the final result generated by the C++ code with the MATLAB script.

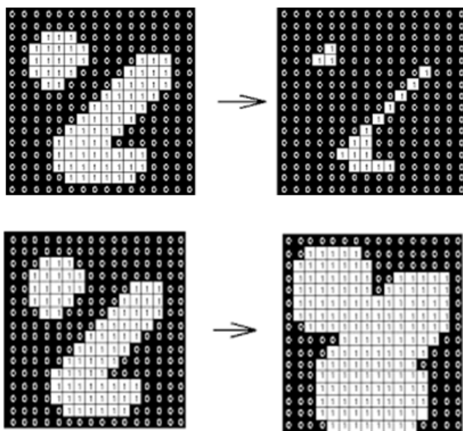

Figure 3. Erosion and Dilation

Figure 3 shows the erosion and dilation filters respectively [3] and [4]. Erosion assigns the min value of the neighborhood to that pixel, while dilation assigns the max value of the neighborhood to the pixel.

Figure 4. Gamma Correction

Figure 4 shows the Gamma correction for an image [5], which is used to change the brightness or luminance of the image according to the following formula and a factor $Y = 0.6$.

$$OM = round\left(\left(\frac{IM}{256}\right)^{\gamma} \times 256\right)$$

The operations implemented are convoluted with the kernels shown below.

0	0	1	0	0
0	1	1	1	0
1	1	1	1	1
0	1	1	1	0
0	0	1	0	0

Figure 5. Erosion and Dilation Kernel

-3	-3	6
-3	6	-3
4	-5	1

Figure 6. Edge Detection Kernel

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

Figure 7. 3x3 Average Blur Kernel

1/25	1/25	1/25	1/25	1/25
1/25	1/25	1/25	1/25	1/25
1/25	1/25	1/25	1/25	1/25
1/25	1/25	1/25	1/25	1/25
1/25	1/25	1/25	1/25	1/25

Figure 8. 5x5 Average Blur Kernel

1	4	6	4	1
4	16	24	16	4
6	24	36	24	6
4	16	24	16	4
1	4	6	4	1

Figure 9. Gaussian Blur Kernel

The algorithm used for the different image processing operations is implemented according to the following flowchart logic.

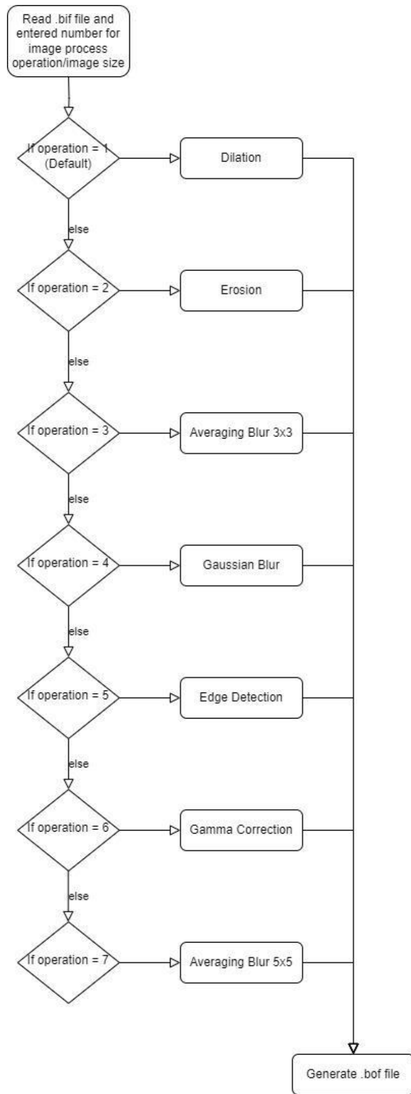

Figure 10. Flowchart

Figure 10 shows the logic flowchart used for the implementation. A binary image file (.bif) is generated by a desired .jpg from the MATLAB script, then the C++ code reads the .bif, and gets the input operation number and desired image size and selects the corresponding operation. A function is then called utilizing the parallel_for TBB implementation as well as a function that utilizes the sequential implementation to compare the processing time. A binary output file (.bof) is then generated as the final result, which then is used in the MATLAB script for further verification.

B. *Parallelization Strategy*

For the parallelization strategy, parallel_for loop is used since an operation has to be performed on each individual pixel, which means multiple pixels could have the same task performed in parallel. It is ideal since every iteration is independent, the number of iterations is known due to the dimensions of the image being known in advance, and every computation depends particularly on the number of iterations performed and the input data uses the iteration count as an index for the operation. The pthreads approach is not used due to the program functioning with multiple image sizes, and TBB would take care of assigning the number of threads in a way that utilize the processor resources more efficiently.

Parallel_for is implemented using a compact lambda expression [6] which takes the inputs of the convolution function and calculates the dot product between the image pixels and the kernel in parallel. This is different than the sequential approach which calculates the dot product one by one for each pixel in scope making the calculation time significantly longer, especially as the size of the image increases. A different parallel_for approach to parallelize the inputs and the kernel before doing the dot product was considered but it created more overhead, which is why the parallel_for was implemented for the dot product only in this project.

```
double conv2D_tbb (IMG_args convo_data)
{
   // it computes convolution (the centered part)
   // It performs the same job as MATLAB conv2(I,K,'same')
   int sX, sY, kX, kY;
   double *I, *K, *O; // these are provided as linear arrays

   I = convo_data.I2; O = convo_data.O2; sX = convo_data.SX; sY = convo_data.SY;
   K = convo_data.K2; kX = convo_data.KX; kY = convo_data.KY;

   // check validity of parameters
   if (!I || !O || !K) return 0;
   if (sX <= 0 || sY <= 0 || kX <= 0 || kY <= 0) return 0;

  tbb::parallel_for (int(0), int(sY), [&] (int i) { // 0 <= i <= SY-1
        tbb::parallel_for (int(0), int(sX), [&] (int j) { // 0 <= j <= SX-1
            conv2DI(convo_data,i,j);
            } );
      } );

  return 1;
}
```

Figure 11. conv2D_tbb Function

As seen in figure 11, the compact lambda expression is used in two parallel_for loops to calculate the dot product in parallel.

```
int conv2DI (IMG_args convo_data, int i, int j)
{
   // it computes convolution (the centered part)
   // It performs the same job as MATLAB conv2(I,K,'same')
   int sX, sY, kX, kY;
   double *I, *K, *O; // these are provided as linear arrays

   I = convo_data.I2; O = convo_data.O2; sX = convo_data.SX; sY = convo_data.SY;
   K = convo_data.K2; kX = convo_data.KX; kY = convo_data.KY;

   // convolution 2d, but provided as linear array.
   int m, n, mm, nn;
   int kCX, kCY; // center index of kernel
   double sum; // temp accumulation buffer
   int i_I, j_I;

   // check validity of parameters
   if (!I || !O || !K) return 0;
   if (sX <= 0 || sY <= 0) return 0;

   // find center position of kernel (half of kernel size, flooring)
   kCX = kX / 2; kCY = kY / 2;

    sum = 0.0; // init to 0 before sum
    for (m = 0; m < kY; m++) // Kernel: rows
    {
        mm = kY - 1 - m; // Flipped kernel: index for rows
        for (n = 0; n < kX; n++) // Kernel: columns
        {
            nn = kX - 1 - n; // Flipped kernel: index for columns
            i_I = i + (kCY - mm);
            j_I = j + (kCX - nn);

            // out-of-bounds input samples are set to 0
            if (i_I >= 0 && i_I < sY && j_I >= 0 && j_I < sX)
                sum += (double) I[sX * i_I + j_I] * K[kX * mm + nn];
        }
    }
    O[sX*i + j] = sum;
  return 1;
}
```

Figure 12. conv2DI Function

Figure 12 shows the convolution function utilized in the parallel_for compcat lambda expression shown in figure 11.

III. Experimental Setup

The project is implemented on three different devices that have different processors: Intel Atom Dual Core Processor N2600 @1.6GHz (Terasic DE2i-150 board), Intel Core i5-6200U 2-Core Processor @2.30GHz, and AMD Ryzen 9 5900X 12-Core Processor CPU @3.70GHz.

A C++ function is used to output and compare the processing time results for the sequential and parallel implementation on the terminal, verify the expected outcome that the parallel approach is faster than the sequential one, and to generate .bof files that are used for the MATLAB script.

The MATLAB script generates the .bif files that are needed for the C++ code. It also includes an implementation of the different image processing operations mentioned, which are then compared to the .bof output files generated from the C++ code to output the different image results and to confirm and verify successful implementation.

IV. Results

The processing time data for each of the different operations and the different image sizes were collected to verify that the TBB parallel approach was faster than the sequential approach.

```
albarazi@DESKTOP-F0H7R8M: ~/project/Final/Final
albarazi@DESKTOP-F0H7R8M:~/project/Final/Final$ ./imgProcess
Warning: Usage: ./imgProcess <modifier1 (1-7)>  <modifier2 (1-5)>
modifier1: 1-Dilation, 2-Erosion, 3-Blur 3x3, 4-Gaussian Blur, 5-Edge Detection, 6-Gamma, 7-Blur 5x5
modifier2: 1-480x307, 2-640x410, 3-940x602, 4-1280x820, 5-1920x1230
Using modifier1 = 1 (dilation) and modifier2 = 3 (940x602) as default
(read_binfile) Input binary file 'house3.bif': # of elements read = 565880
(read_binfile) Size of each element: 1 bytes
(write_binfile) Output binary file 'house_d_3.bof': # of elements written = 565880
(read_binfile) Size of each element: 1 bytes
Sequential Approach:
        start: 345269 us        end: 378027 us
        Elapsed time (actual computation): 32758 us
TBB Approach:
        start: 434865 us        end: 490105 us
        Elapsed time (actual computation): 55240 us
albarazi@DESKTOP-F0H7R8M:~/project/Final/Final$ ./imgProcess 4 3
(read_binfile) Size of each element: 1 bytes
(read_binfile) Input binary file: # of elements read = 565880
(write_binfile) Output binary file 'house_gb_3.bof': # of elements written = 565880
(read_binfile) Size of each element: 4 bytes
Sequential Approach:
        start: 520499 us        end: 571031 us
        Elapsed time (actual computation): 50532 us
TBB Approach:
        start: 599330 us        end: 622911 us
        Elapsed time (actual computation): 23581 us
albarazi@DESKTOP-F0H7R8M:~/project/Final/Final$
```

Figure 13. Output on Terminal

Figure 13 shows the processing time output on the terminal for both the sequential and TBB approaches. The first run shows the execution command with one argument "imgProcess", which warns the user of the modifiers that can be used, and assigns default values for the modifiers, which are chosen to be 1 for modifier 1, and 3 for modifier 2. Modifier 1 is used for the operation number, while modifier 2 is used for the image size. The second run shows the command with the input users for both modifiers. The results are then tabulated accordingly.

480 x 307 Image (147,360 elements)

Table 1 - Intel Atom Dual Core Processor N2600 @1.6GHz (Terasic DE2i-150 board)

Operation	Dilation	Erosion	Gamma	Edge Detection	3x3 Blur	5x5 Blur	Gaussian Blur
Sequential (µs)	51700	50551	85983	45041	36565	60768	69335
TBB (µs)	39357	48011	66159	36504	28333	49396	50005

Table 2 - Intel Core i5-6200U 2-Core Processor @2.30GHz

Operation	Dilation	Erosion	Gamma	Edge Detection	3x3 Blur	5x5 Blur	Gaussian Blur
Sequential (µs)	12331	7470	10197	6360	6149	9809	11508
TBB (µs)	10618	5848	5017	5554	5710	7564	8799

Table 3 - AMD Ryzen 9 5900X 12-Core Processor CPU @3.70GHz

Operation	Dilation	Erosion	Gamma	Edge Detection	3x3 Blur	5x5 Blur	Gaussian Blur
Sequential (µs)	2266	3208	1624	1917	1859	2610	2762
TBB (µs)	2069	2288	1371	1685	1465	2120	2161

640 x 410 Image (565,880 elements)

Table 4 - Intel Atom Dual Core Processor N2600 @1.6GHz (Terasic DE2i-150 board)

Operation	Dilation	Erosion	Gamma	Edge Detection	3x3 Blur	5x5 Blur	Gaussian Blur
Sequential (µs)	90465	85517	146165	74248	75777	100853	110525
TBB (µs)	78079	66223	106778	63271	64053	98834	103006

Table 5 - Intel Core i5-6200U 2-Core Processor @2.30GHz

Operation	Dilation	Erosion	Gamma	Edge Detection	3x3 Blur	5x5 Blur	Gaussian Blur
Sequential (µs)	14426	16097	11604	12222	10274	17111	16025
TBB (µs)	9836	11911	9025	8333	7716	12507	12567

Table 6 - AMD Ryzen 9 5900X 12-Core Processor CPU @3.70GHz

Operation	Dilation	Erosion	Gamma	Edge Detection	3x3 Blur	5x5 Blur	Gaussian Blur
Sequential (µs)	4004	5659	2800	3025	2470	4727	4820
TBB (µs)	2319	2363	2337	2596	2266	2265	2409

940 x 602 Image (565,880 elements)

Table 7 - Intel Atom Dual Core Processor N2600 @1.6GHz (Terasic DE2i-150 board)

Operation	Dilation	Erosion	Gamma	Edge Detection	3x3 Blur	5x5 Blur	Gaussian Blur
Sequential (µs)	177845	191801	328897	140795	147901	243157	283595
TBB (µs)	128292	120701	192432	107801	114298	178997	200896

Table 8 - Intel Core i5-6200U 2-Core Processor @2.30GHz

Operation	Dilation	Erosion	Gamma	Edge Detection	3x3 Blur	5x5 Blur	Gaussian Blur
Sequential (µs)	27416	30031	17240	17475	17134	28852	33812
TBB (µs)	19353	17440	10458	15182	14148	24364	27536

Table 9 - AMD Ryzen 9 5900X 12-Core Processor CPU @3.70GHz

Operation	Dilation	Erosion	Gamma	Edge Detection	3x3 Blur	5x5 Blur	Gaussian Blur
Sequential (µs)	8518	12086	6033	6235	4992	9565	10336
TBB (µs)	3057	3572	2630	4437	2939	3542	3631

1280 x 820 Image (1,049,600 elements)

Table 10 - Intel Atom Dual Core Processor N2600 @1.6GHz (Terasic DE2i-150 board)

Operation	Dilation	Erosion	Gamma	Edge Detection	3x3 Blur	5x5 Blur	Gaussian Blur
Sequential (µs)	349611	319996	581107	273305	260267	410755	398574
TBB (µs)	158223	155130	344743	250129	256296	260917	297251

Table 11 - Intel Core i5-6200U 2-Core Processor @2.30GHz

Operation	Dilation	Erosion	Gamma	Edge Detection	3x3 Blur	5x5 Blur	Gaussian Blur
Sequential (µs)	53153	57552	38139	34844	36914	57270	55719
TBB (µs)	31935	45404	19288	29905	26942	46209	45081

Table 12 - AMD Ryzen 9 5900X 12-Core Processor CPU @3.70GHz

Operation	Dilation	Erosion	Gamma	Edge Detection	3x3 Blur	5x5 Blur	Gaussian Blur
Sequential (µs)	15635	22347	11271	10946	8898	17344	17437
TBB (µs)	4365	4644	3171	5837	4034	4753	5325

1920 x 1230 Image (2,361,600 elements)

Table 13 - Intel Atom Dual Core Processor N2600 @1.6GHz (Terasic DE2i-150 board)

Operation	Dilation	Erosion	Gamma	Edge Detection	3x3 Blur	5x5 Blur	Gaussian Blur
Sequential (µs)	745256	697655	1207770	523887	515310	908358	939032
TBB (µs)	355341	357881	524633	443125	416930	595624	625761

Table 14 - Intel Core i5-6200U 2-Core Processor @2.30GHz

Operation	Dilation	Erosion	Gamma	Edge Detection	3x3 Blur	5x5 Blur	Gaussian Blur
Sequential (µs)	112791	140199	74927	70594	71998	127611	140309
TBB (µs)	64426	82672	50029	54101	52634	94642	109570

Table 15 - AMD Ryzen 9 5900X 12-Core Processor CPU @3.70GHz

Operation	Dilation	Erosion	Gamma	Edge Detection	3x3 Blur	5x5 Blur	Gaussian Blur
Sequential (µs)	35190	50627	25496	25992	20943	37405	41061
TBB (µs)	7074	7298	52634	5076	6847	8619	8852

As seen from the data collected from tables 1-15, TBB beats sequential in speed in all cases. As the filter becomes more complex, the calculation time increases. As seen in table 1, the time for the 5x5 blur is much higher than the 3x3 kernel, however, the TBB implementation decreases the calculation time. As seen in table 13, as the image size increases, so does the calculation time of the operation. This can be seen between tables 7 and 10, where the time of the operation is roughly doubled due to the amount of input elements being doubled. However, as seen in table 13, the ratio between the speed of the TBB implementation vs the sequential increases tremendously being more than twice as fast. Finally, another observation after testing on multiple devices is that, as the processor core count and speed increase, the time it takes for both the sequential and TBB calculations decrease significantly, however the difference between them is much higher as the TBB becomes more efficient.

Figure 14. Original Image

Figure 15. Greyscale Image

Figure 16. Dilation

Figure 17. Erosion

Figure 18. Gamma

Figure 19. Edge Detection

Figure 20. 3x3 Averaging Blur

Figure 21. 5x5 Averaging Blur

Figure 22. Gaussian Blur

Figure 14 shows the original input image that was used to implement the operations on [7]. Figure 15 shows the grayscale image generated in MATLAB. Figures 16 – 22 show the different image processing operations applied, which are generated in MATLAB using the output .bof files from the C++ code and compared to the images generated using the MATLAB functions.

Results conclude that TBB is faster/better than sequential in image processing operations. As image size or number of elements increase, processing time increases and TBB becomes much more efficient than sequential. As the count of the process cores and speed increase, the computation time for the operation decreases.

CONCLUSIONS

In summary, parallel programming is a much more efficient approach than sequential for image processing. Different factors such as the image size or number of elements, different processors' core count and speed play a significant role in the operation calculation time.

Future improvements include additional image processing operations and performing multiple operations consecutively to do more complex applications such as object detection which would utilize multiple different kernels. A MATLAB script could then be improved to convert a .bof file generated from the C++ code back to a .bif file that the code can use again to perform the next operation. To improve the parallel strategy, a parallel pipeline could be used to assist in indexing the input pixels concurrently, which would not speed up the calculation time of the operation itself, but could improve the loading time for very large images. Finally, a Graphical User Interface (GUI) can be implemented for a more user-friendly image processing tool.

REFERENCES

[1] Scott Rome, "Why blurring an image is similar to warming your coffee," Scott Rome, https://srome.github.io/Why-Blurring-an-Image-is-Similar-to-Warming-Your-Coffee/ (accessed Dec. 10, 2023).

[2] Edge detection of image using opencv (CV2) in Python, https://www.includehelp.com/python/edge-detection-of-image-using-opencv-cv2.aspx (accessed Dec. 10, 2023).

[3] "Erosion," Morphology - Erosion, https://homepages.inf.ed.ac.uk/rbf/HIPR2/erode.htm (accessed Dec. 10, 2023).

[4] "Dilation," Morphology - Dilation, https://homepages.inf.ed.ac.uk/rbf/HIPR2/dilate.htm (accessed Dec. 10, 2023).

[5] "Imadjust," MathWorks, https://www.mathworks.com/help/images/gamma-correction.html (accessed Dec. 10, 2023).

[6] D. L. Obregon, TBB: parallel_for, Tutorial 5, https://www.secs.oakland.edu/~llamocca/Tutorials/Emb_Intel/Tutorial%20-%20Unit%205.pdf (accessed Nov. 15, 2023).

[7] "New Deck for chanhassen couple," Iron River Construction, https://ironriverco.com/featured-projects/new-deck-gives-chanhassen-couple-space-to-host-outdoors/ (accessed Dec. 10, 2023).