Abstract—The capability to detect edges in an image is a major component in the field of image processing. That being said one of the most commonly utilized methods for edge detection is the Canny edge detection algorithm. In this paper we outline and define what edge detection is in image processing, and how the Canny edge detector works in typical implementations. We briefly refer to other papers which have similarly looked into optimizing the Canny edge detector and then propose our own hypothesis on how to parallelize this algorithm via multithreading. Our current code implementation is then explained alongside current results and issues.

Keywords—Canny Edge detection, parallel, multithreading, Robot Vision, image processing.

I. INTRODUCTION

In image processing edge detection refers to the process of seeking out and identifying the points or pixels in an image scene at which the brightness between adjacent pixels changes sharply. Typically if implemented well edge detection allows us to see line patterns where certain features are present in an image. Hence why edge detection is commonly used in applications such as facial recognition, object recognition and image analysis which require the capability to identify specific features in a scene. Edge detection is also used in some cases to detect depth and distance of objects in an image.

The capability to detect edges allows those in the fields of image processing and robot vision to implement a wide variety of applications including those mentioned previously. It is because of this that a lot of research has been dedicated to finding optimal methods to detect edges. The Canny edge detector is perhaps one of the most widely utilized of such methods having been implemented in a variety of ways since its development in 1986 by John F. Canny. The issue however is that in the nearly thirty years that have since passed images have multiplied in size and resolution capabilities. Because of this the Canny algorithm can often be time consuming and is computationally inefficient in processing larger modern images.

II. RELATED WORKS

Researchers have come up with a few ways to optimize the Canny edge detector however we have seen limited research on improving the Canny Operator solely through multithreading means to take advantage of multiple computer cores using concurrent containers and libraries.[1], [2], [3], [4] Most optimizations involve small CPU optimizations, the use of more efficient equations (see S. Sakar and K. Boyer [5]) or purely GPU implemented optimizations.[6]

For example, C. Gentsos, C. Sotiropoulo and S. Nikolaidis [7] have implemented a parallel optimization for the canny edge detector however this implementation only works on field programmable arrays and requires some knowledge of hardware engineering. Another group of researchers S. Niu, J. Yang, S. Wang and G. Chen have implemented parallel optimization through the Graphic Processing Unit and its architecture (CUDA). While this method is highly successful it only focusses on GPU optimizations rather than CPU optimizations which we would like to employ. The same applies to researchers J. Fung, S. Mann and C. Aimone [8] who also took the approach of implementing GPU optimizations.

Our current implementation of the Canny edge detector takes advantage of multithreading and uses multiple threads to scan several areas of an image simultaneously for edges. There are several concurrent data structures[9], [10], [11] that are built to support multithreading. These data structures are extensively used in multiple areas of engineering and computer science.[12], [4] The threads then each take their scanned portions and combine them together to produce the resulting edge detected image. The ultimate goal is to have the Canny edge detector process an image at a fraction of its typical implementation. Current results appear to be promising despite their being a few issues which will be discussed in the results section of this paper.

III. HOW CANNY EDGE DETECTION WORKS

The following is a general explanation of how the Canny edge detection process works. This consists of a three step process which involves applying a smoothening algorithm onto the image, then finding the intensity peaks in the image and finally applying a double threshold onto the image before finally outputting the results of these steps.

A. Taking an input and Applying a smoothenner

To start off the Edge detector takes a single input image. An image in terms of Robot Vision is essentially a two dimensional array of integer values. Each value represents the shade or pixel intensity value. For grayscale images each array slot is a value representing the shade of the pixel. Values
Typically Canny algorithms first implement a smoothening technique onto input images. What this does is it removes any unwanted pixel intensity deviations (referred to as image noise) which might otherwise throw off the algorithm. A Gaussian smoothening is usually performed by using two Gaussian masks which iterate through the image in the x and y directions and perform a convolution which calculates the weighted sum of a small neighborhood of pixels. These masks may be of any size but smaller 3 by 3 or 5 by 5 masks are usually implemented because they have a higher sensitivity in finding and removing image noise.

We go ahead and use the Sobel operator for our Canny Implementation since it is a commonly used one. We apply two Sobel masks, but rather than use the original Sobel masks (Fig 2) we implement a Gaussian mask onto the image. We then take the two output tables produced by the masks and use the x and y values of each pixel to find its magnitude $G = \sqrt{G_x^2 + G_y^2}$ We then search through all of the magnitude values and identify all pixels which pass an intensity threshold. All other pixels are ignored.

The goal of the smoothening is to create a much clearer definition of where edges are present in an image. Fig.3 shows the original output of edges detected from an input image followed by the smoothened version of the output.

### B. Finding the Intensity Peaks

The next step involves iterating through all of the pixels in an image in order to identify peaks of intensity in the image. A peak pixel is defined as a pixel whose brightness is greater than its adjacent neighbors. For each pixel a maximum peak test is employed. In order to do this the slope or direction of a pixels gradient is taken into account. Knowing this allows us to test if the pixel is a peak against its adjacent pixels in the proper directions. Those pixels who are found to be peaks are marked as candidate pixels which must be tested before becoming part of our final output.

#### C. Double Thresholding

In order to test the candidate pixels a histogram is created with the pixel intensity values in the input image. Based off of the histogram a high and low threshold are applied to the map of magnitude values which has been created for the image.

All pixels whose magnitudes are above the High threshold are accepted into the final image, all pixels whose values are below the low threshold are dismissed and all those in between both thresholds are accepted only if they have a neighboring peak/pixel which is above the high threshold. The final output image is then produced containing all of the marked pixels which passed the double threshold algorithm.

### IV. Our Implementation

When implementing the Canny edge detector we took into account that not all images are in grayscale. In the case of a color image each pixel or array slot contains a set of values representing the Red, Green and Blue (RGB) shades present in the colored pixel. In order for the Canny edge detection to work we take the input image and convert it into a grayscale image by utilizing features of the Java language. This allows us to detect jumps in pixel brightness with much more ease because we are now only concerned with the grayscale values.
of each pixel rather than having to contend with the three Red, Green and Blue shades.

A. Comparison to the General Implementation

Before proceeding some explanation is necessary in order to explain how we have implemented the Canny Algorithm as opposed to the general implementation.

As mentioned above the first step in Canny Edge detection is to smooth an image. Two of the commonly utilized smoothening methods are the Sobel and Prewitt methods. We opted to apply a Sobel Gaussian operator as stated previously. The reason for this being that this is one of the most commonly utilized and efficient methods, however this implementation can be easily changed to utilize other smoothening methods. On a similar note it should be noted that our peak detection and double threshold steps also work in much the same way as the typical Canny implementation.

B. Multithreading

Our idea is to optimize the Canny edge detector by implementing multithreading into the process. The goal here is to make use of multiple computer cores in order to improve the run time/performance of the Canny algorithm. [14], [15] Do this by having multiple threads divide the input image into several smaller sections. Each thread will then simultaneously run the Canny algorithm on its acquired section so that the edges are detected in a fraction of time proportional to the number of threads running the program.

We have thus implemented the canny edge detecting code in three separate ways. The first implementation runs the Canny algorithm sequentially with only a single thread. This code is used as a controlled comparison against the other two codes which are different only in that they implement multiple threads into the edge detection process.

The second implementation runs the Canny Edge detector with two threads. To work with this the original input image into two halves which will each in turn become input images for each thread. In this implementation each thread takes one half of the input image, the first thread takes the top half, the second takes the bottom half. Each thread then proceeds to process their respective input with the Canny algorithm. Similarly the third implementation doubles the number of threads to four. In this case each the image is divided into four quarters.

C. Technicalities in our Implementation

Initially the prospect of equally dividing an image into portions depending on the number of threads would seem to be enough to have a flawless multithreaded canny edge detector however this is not the case.

For example, initially when creating the two threaded implementation we ran into a few issues with the final output of the joined threads. Since each thread takes half of the original input image this creates a divide in the middle of the image where the split has been made for the threads to take their portions. The result was that this dividing line was then omitted from the edge detection process. When the two picture halves are then recombined this then leaves a blank one pixel long line across the final outputted image. This can be observed by the arrow pointing to a division line in Fig.4 which was the output of the initial two threaded process. Without major scrutiny this did not appear to pose a major problem because the division line is rather small and not immediately noticeable, furthermore the output was otherwise accurate.

The greater issue that arose because of the division problem is that the more threads that we would attempt to implement in our process the more dividing lines we would have in our results. With four threads for example we would have two dividing lines one across the horizontal axis and another across the vertical axis. So although the code was indeed successfully running in a multithreaded fashion and would result in decreased processing time as more threads were added it would also result in more flaws as more threads were added.

Fig. 4. Figure 4: Labeled above is an arrow pointing to a division line.

In order to combat this problem we opted to slightly alter the way by which the original input image was divided. Rather than have equally divided portions some threads take on an extra pixel line or two as a buffer to prevent division lines from appearing in the final output. Because dividing lines are only a one pixel long line across the image adding a few pixel lines to some threads does not create any noticeable difference in the runtime of the multithreaded implementations.

Division of image portions in our implementation occurs as follows. Note that the following explanation is merely in pseudo code. An input image Img is accepted by the code.
We then take the "height = Img.getHeight();" and the "width = Img.getWidth();". The image is converted into a Grayscale image utilizing java features "GrayImg = Img.toGrayScale".

We then opted to divide the image into horizontal portions so as to not have to contend with vertical division lines. This allows us to perform less corrections to compensate for the extra division lines when adding more threads. Each thread minus the last one will then take an equal portion of the image plus an extra two lines to compensate for the division line between itself and the below image portion taken by the following thread.

For our dual threaded implementation for example we can describe thread one with pseudo code as $T1 = (0, Img.width, (Img.height/2) + 2)$ while thread two would be $T2 = ((Img.height)/2, Img.width, (Img.height)/2)$, where the first given value within the parenthesis is the starting height value of the given image portion, the second value represents the width of the image and the third value represents the ending height value of the given image portion.

Of note however is that in implementing this correction we have to pay special attention to how we add extra lines to threads as we scale the code to work with larger numbers of threads. Although we did not implement the canny edge detector to work with a large quantity of threads it can be easily observed that more corrections would need to be performed for larger numbers. This can be easily adjusted with the addition of a few lines of code for different thread numbers.

Current results are promising and are discussed in greater detail in the following section. The code implemented has been proven to work within a proper degree of accuracy with computers containing four cores.

V. RESULTS AND CONCLUSIONS

A. Testing Requirements

As mentioned previously three separate code implementations were developed. One is a sequential implementation while the other two run with two and four threads respectively. The reasoning behind this is that as a team we had access to dual and quad core computers at best. Should one want to develop further optimizations to the code to work with eight threads and above this can easily be done as previously mentioned with minimal changes to our current multithreaded implementations.

In order to test the successful implementation of the multithreaded canny algorithm several factors are taken into account. Firstly a code implementation which makes use of more threads than another implementation must have a noticeably shorter runtime in order to be truly successful. Second each corresponding thread must correctly detect edges in its given image portion. Finally the final output image should correctly represent the edge detected version of the original input image with minimal artifacts and image noise present in the image.

Testing the first of the above mentioned requirements was done by timing the execution of each code implementation from the time that the input image is accepted until the final output image is produced. We tested our implementations on several dual and quad core computers several times in order to develop average run times for each implementation.

In order to ensure the second requirement we observe the output image portions which each thread produces after running the Canny algorithm. These image portions become byproduct outputs of our full code implementations and can be observed in the file source destination of the code.

Finally the third requirement is ensured by running all three code implementations with the same input image and then comparing the outputs of the multithreaded implementations against the sequential implementation. Ideally the multithreaded implementations should appear to have identical outputs to the sequential implementation.

B. Results

Results of our two multithreaded implementation seem promising. Each thread is able to perform the Canny Edge Detection simultaneously without issue on their corresponding image section for both multithreaded implementations. The final output of the multithreaded implementations is nearly identical to that of the sequential implementation with the exception of some artifacts which appear in output images, however these artifacts are usually minors thus the resulting output images are well within our accepted parameters.

In terms of run time our results indicate that we have been successful in implementing the canny algorithm within a fraction of the sequential time. For a small image of approximately 550 by 600 pixels the sequential implementation takes an average of 2.19 seconds to run while the dual threaded implementation takes an average of 0.430 seconds, and furthermore the quad threaded implementation takes an average of .370 seconds.

For an image of approximately 1200 by 1600 pixels in size the sequential algorithm takes an average of 4.55 seconds to run. The dual threaded implementation takes approximately 2.11 seconds and the quad threaded implementation takes 1.71 seconds.

C. Image Results and examples

D. Conclusions

We can indeed observe from these results that there is a vast improvement in runtime between the sequential and dual threaded implementations. The quad threaded implementation is indeed an improvement to the dual threaded implementation although not by the same degree of improvement observed in comparing the first two implementations. At the same time we can also observe some minor artifacts which occur in the quad threaded implementation in comparison to the output of the sequential implementation. This may be due to the fact that the division of images results in a reduced acceptance of intermediate candidate pixel peaks. As stated above these are peaks which in order to pass the double threshold test must have neighboring pixels which have already been accepted into the final output. The division of the image into portions will reduce the number of neighboring pixels along division lines and will reduce the number of neighboring intermediate pixels...
accepted. Though this is an unintended side effect of applying multithreading into the algorithm the results are still highly acceptable as the vast majority of peaks are still easily detected by the Canny algorithm.

Further testing on computers with greater numbers of cores would be required in order to compare these results to implementations with eight or more threads. Other methods of utilizing multithreading in the canny edge detection process may also be possible and may prove to further optimize the algorithm such as developing a method to parallelize the initial smoothening process or the peak intensity pixel detection process. C. Gentsos, C. Sotiropoulos and S. Nikolaidis [7] have performed similar optimizations albeit with the need for specific hardware changes and requirements.

ACKNOWLEDGMENT

We would like to thank Professor Dechev for having a great class!

REFERENCES


