Parallel Implementation of Canny algorithm using Intel TBB

Sonal Nimje
Department of Computer Science
Oakland University
Email: sonalnimje@oakland.edu

Weiting Peng
Department of Computer Science
Oakland University
Email: weitingpeng@oakland.edu

Abstract—**The modern development in hardware and design technologies have made multi-core processors a dominant market trend in desktop computers and high end mobile devices. To get maximum performance from modern multi-core systems, it is necessary to structure an application to use as many cores as possible. One popular method to do this is data parallel programming. In this model, the programmer describes a computation to apply in parallel across an entire data set at the same time, operating on independent data elements. Over recent years, many run-time systems have been developed to achieve this. Rather than having programmers divide work among threads, the run-time system decides how to map the parallel work across the hardware's processors. Multimedia streams e.g. video often have large amounts of data on which similar operations are repeatedly performed, so data parallel programming is frequently used for media or image processing. In this paper, we will evaluate data parallelism using run-time Intel library 'Intel Threading Building Blocks'. Canny operator is the widely used technique for edge detection in image processing. We will implement Canny edge detector using sequential and data parallel method using Intel TBB. For performance evaluation we will measure the run-time of sequential and Intel TBB method. Further we will check the performance of Canny algorithm using Intel TBB on the systems with different number of cores.**

Keywords—**multi-core; Intel Threading Building Block (TBB); Canny algorithm, data parallelism.**

1. Introduction

Now a days multiple core CPUs are available in all modern computers and has become a necessity for increasing the computing power. We have reached a limit where we cannot increase the clock frequency of the processors due to various constraints like heat and power. To gain the advantage of increased processing power from these multiple core CPUs we need to write programs to be able to use these multiple cores and process the task in parallel on each core and reduce the overall execution time. Intel has developed a platform independent library to meet this purpose.

Intel Threading Building Blocks (Intel TBB) is a run-time based parallel programming model for C++ code that uses threads. It consists of a template-based run-time library to help you harness the latent performance of multi-core processors. Intel Threading Building Blocks can be used to write scalable applications that:

- Specify logical parallel structure instead of threads
- Emphasize data parallel programming
- Take advantage of concurrent collections and parallel algorithms

Intel TBB is language and compiler independent. It is mainly used for data parallel programming. It supports nested parallelism, so you can build larger parallel components from smaller parallel components. User specifies tasks, not threads and let the library map tasks onto threads in an efficient manner.

Intel TBB emphasizes data-parallel programming, where multiple threads work on different parts of data. Using data-parallel programming, we can increase the scalability by dividing large data into smaller parts. With this approach program performance increases as number of processors.

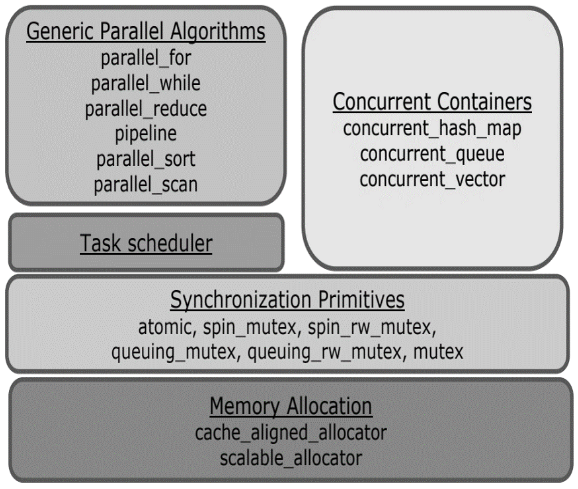


Figure 1. *Intel TBB Components*

Canny edge detector is most reliable detection methods in image processing algorithms.

In this paper, we will report our implementation of this algorithm and the speedup achieved over sequential method.

We will implement this algorithm on Intel multi-core architecture by exploiting the potential parallelism provided by the multi-cores in a single chip.

We will experiment the possible performance improvement on the processing time by comparing the run-time of sequential and parallel implementation using Intel TBB.

2. PAST RESEARCH

In 2014 research paper “Parallelizing Doolittle Algorithm Using TBB”, published by Sushil Shah and Dinesh Naik [1], a different approach applied for paralleling Doolittle algorithm with the help of Intel TBB.

The process was divided in three parts:

Decomposing the data, processing the data in parallel, finally composing the data. They divided the large matrix data into sub-matrices of data and gave these matrices to different cores for processing using Intel TBB. They also applied task parallelism where different operations can be running on the same data or may be working on different data or a combination of both. Results show that for matrices of order less than 100, the running time of parallel algorithm is slightly higher than that of sequential because the overhead involved in using threads on different cores of CPU. For matrices of order higher than 100, the running time of parallel algorithm is always lesser than sequential and as the order of matrix increases the difference in running time of parallel and sequential algorithm increases.

In 2011 research about “Accelerating Multimedia Applications using Intel Threading Building Blocks on Multi-core Processors” was published by Cheong Ghil Kim [2]. Research was done to accelerate the multimedia performance using Intel TBB. It implemented the Sobel operator to detect edges in the images. They implemented the Sobel operator on different sizes of images and measured the effect of Intel TBB implementation on the Speedup. Steady performance improvement was observed with Intel TBB implementation with the growing size of image.

Both these research show that TBB implementation has improved performance over serial implementation, however results does not show the quantification of scalability of test applications developed. In our research we implemented Canny Edge detector using TBB. We have also measured the scalability achieved, i.e. how test application performs on the systems with different number of cores.

3. Intel TBB Features

Work-stealing task scheduler:

Each thread maintains an (approximate) deque of tasks. A thread performs depth-first execution. Uses own deque as a stack. If thread runs out of work, it steals tasks from other threads, treat victim’s deque as queue. Stolen task tends to be big, and distant from victim’s current effort. [11]

Concurrent containers:

IntelTBB provides concurrent containers. STL containers are not safe under concurrent operations. attempting multiple modifications concurrently could corrupt them. Standard practice would be to wrap a lock around STL container accesses. It limits accessors to operating one at a time, killing scalability. TBB provides fine-grained locking and lockless operations where possible, worse single-thread performance, but better scalability. It can be used with TBB, OpenMP, or native threads [11].

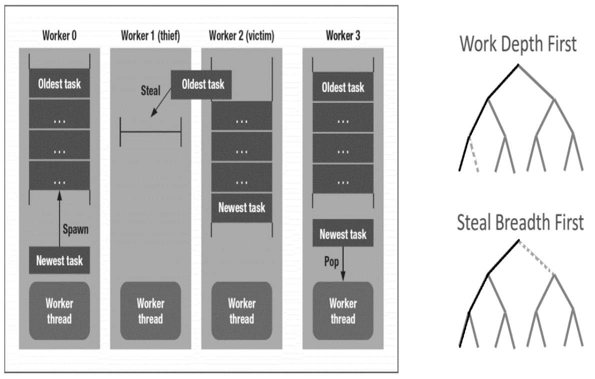


Figure 2. *Work-Stealing task scheduler [11]*

Scalable memory allocator:

Memory allocation is often a bottle-neck a concurrent environment, because Thread allocation from a global heap requires global locks. Intel Threading Building Blocks provides tested, tuned, scalable, per-thread memory allocation [11].

Scalable memory allocator interface can be used:

- As an allocator argument to STL template classes
- As a replacement for malloc/realloc/free calls (C programs)
- As a replacement for new and delete operators (C++ programs)

Low-level synchronization primitives:

Intel Thread Checker can automatically detect race conditions. All TBB mutual exclusion regions are protected by scoped locks [11].
Several mutex behaviors are available:

- Spin vs. queued (“fair”vs. “unfair”). spin_mutex, queuing_mutex
- Multiple reader/ single writer. spin_rw_mutex, queuing_rw_mutex
- Scoped wrapper of native mutual exclusion function. mutex(Windows*: CRITICAL_SECTION, Linux*: pthreads mutex)

3.1. Generic parallel Algorithms

TBB provides several parallel algorithms that help to make common operations parallel for e.g. parallel_for, parallel_sort, parallel_reduce, parallel_scan, parallel_while etc

[11]. Some of the parallel algorithms are described below which we have used in our research.

3.1.1. Iteration space/ Range. To divide the work into chunks that are equally spread on the tasks, a range concept is used. A range object provides functions to recursively split the iteration space into smaller chunks. The size of the chunk is evaluated by the scheduling system.It uses range object t split. Splitting is stopped if the serial execution is faster than to split further [11].

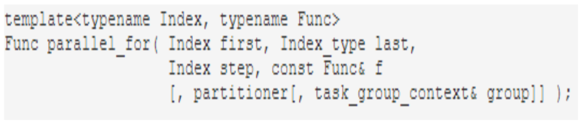
blocked_range<Value>

A blocked_range<Value>represents a half-open range [i,j) that can be recursively split. It is constructed with the start range value i and the end range value j. An optional third parameter defines the grain size which specifies the degree of splitting [6].

blocked_range2d<RowValue,ColValue>

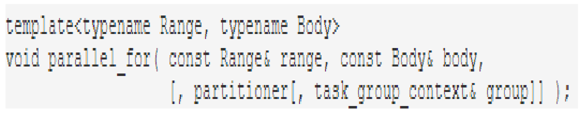
A blocked_range2d<RowValue,ColValue>represents a half-open two dimensional range [i0,j0) x [i1,j1). The grain size can be specified for each axis of the range.Each axis of the range has its own splitting threshold [6].

3.1.2. parallel_for template. The parallel_for function template performs parallel iteration over a range of values. There are two versions for parallel_for, a compact notation and a more general version that uses a range explicitly.



```
template<typename Index, typename Func>
Func parallel_for( Index first, Index_type last,
                   Index step, const Func& f
                   [, partitioner[, task_group_context& group]] );
```

Figure 3. *Syntax for parallel_for compact notation [3]*



```
template<typename Range, typename Body>
void parallel_for( const Range& range, const Body& body,
                   [, partitioner[, task_group_context& group]] );
```

Figure 4. *Syntax for parallel_for general usage [3]*

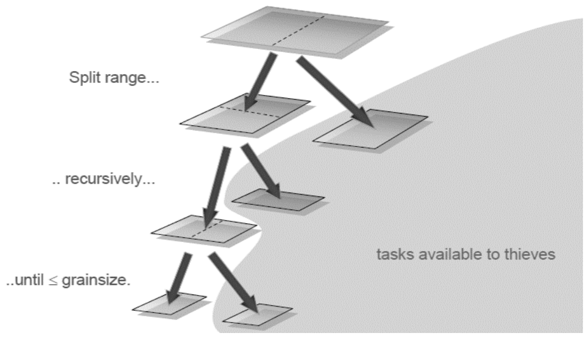


Figure 5. *Recursive splitting in blocked_range2d [11]*

4. CANNY ALGORITHM

Edges in the images characterizes boundaries and therefore carries great importance in image processing. Edges in the images are the areas where there is high contrast in intensity from one pixel to another. Finding edges are important as it filters out the useless information and preserves structural properties in an image.

Canny edge detector is widely used technique in edge detection. It extracts useful structure information from different visual objects and greatly reduces the amount of data to be processed. It has been widely used in various computer vision systems. Canny found that the requirements for edge detection are quite similar in different vision systems. Therefore, a kind of edge detection technology with extensive application significance can be realized. General criteria for edge detection include [4]:

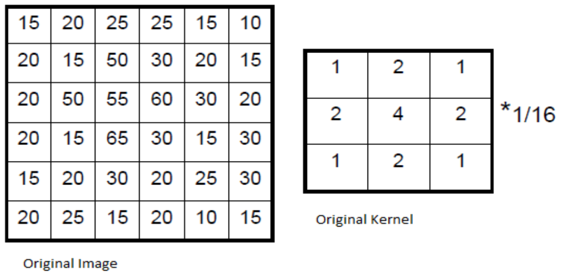
- Edge detection with a low error rate means that as many edges in the image need to be captured as accurately as possible.
- The detected edge should be precisely positioned in the center of the real edge.
- The given edge in the image should only be marked once, and where possible, the noise of the image should not produce false edges.

Canny edge detection algorithm can be divided into the following steps [4]:

4.1. Gaussian filtering to remove noise

Noise can easily affect the edge detection, so first step is to filter out the noise. The Gaussian filter is used to blur and remove unwanted detail and noise. Gaussian filter uses 2D convolution method. Convolution is performed by sliding the kernel or mask over a grey-level image. The kernel starts from the top left corner and moves through entire image within image boundaries. Each kernel position corresponds to a single output pixel. Each pixel is multiplied with the kernel cell value and added together.

4.1.1. Gaussian filtering example:. Standard 3x3 Kernel is used in Gaussian filtering as shown in figure 6.
This Kernel can be broken into 1-D kernel in x-direction



15	20	25	25	15	10
20	15	50	30	20	15
20	50	55	60	30	20
20	15	65	30	15	30
15	20	30	20	25	30
20	25	15	20	10	15

Original Image

1	2	1
2	4	2
1	2	1

*1/16

Original Kernel

Figure 6. *Gaussian Kernel [13]*

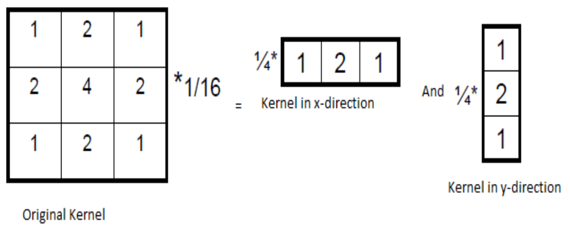


Figure 7. *1-D Gaussian Kernel [13]*

and 1-D kernel in y-direction [13].
 x-direction kernel is applied to original image as shown in figure 8.
y-direction kernel is applied to the first output which gives the final blur image as shown in figure 9.

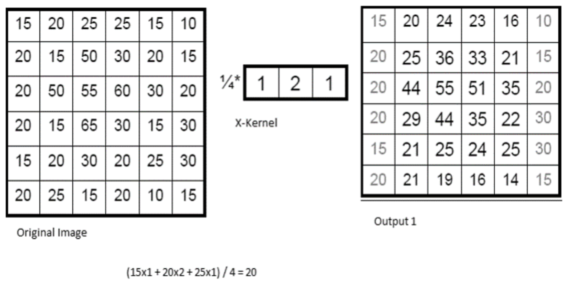


Figure 8. *Output 1 [13]*

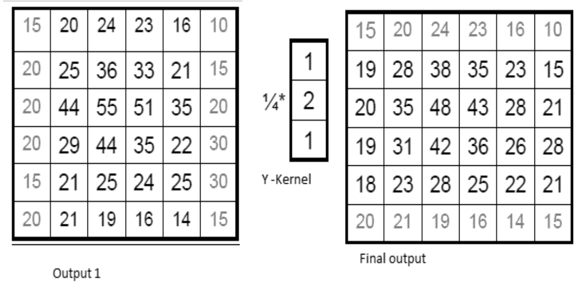
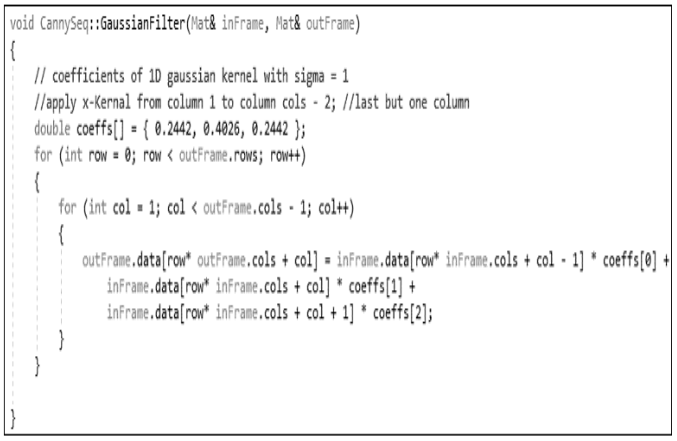


Figure 9. *Final output [13]*

4.1.2. TBB parallel_for use in for loop:. In sequential method first for loop is applied to apply x-kernel on each pixel and 2nd for loop is applied to apply y-kernel in first output.

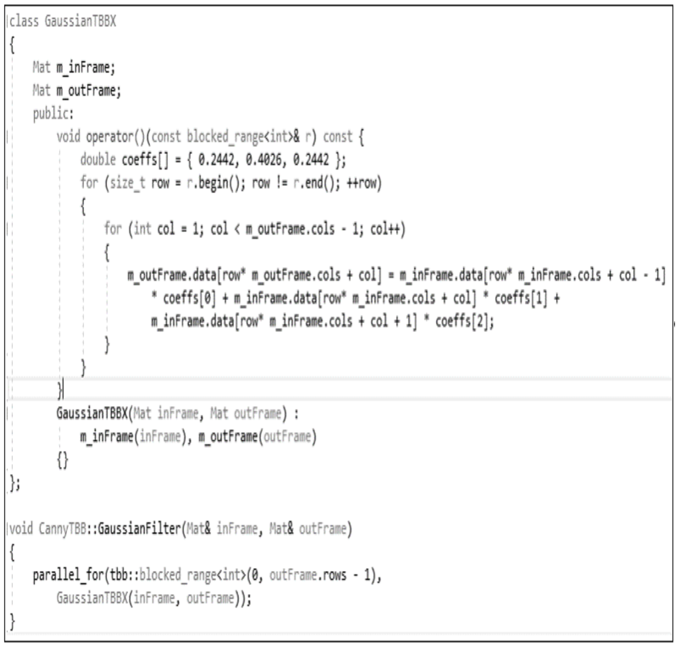
The template function tbb::parallel_for breaks this iteration space into chunks, and runs each chunk on a separate thread. Using parallel_for loop, pixel operations can be divided on separate thread. First parallel_for loop is used for applying x-direction kernel. 2nd parallel_for loop is used for applying y-direction kernel.

The codes shown in figures 10 and 11, show the usage of TBB with Gaussian filtering. The former shows its serial version and the later the corresponding parallel version which uses blocked_range to specify the iteration space. Header starts with #include “tbb/blocked_range.h” statement. The blocked_range enables the outermost loop of the serial version to become parallel loop. The parallel_for recursively splits the blocked_range. It invokes GaussianTBBX::operator() on divided subranges.



```
void CannySeq::GaussianFilter(Mat& inFrame, Mat& outFrame)
{
    // coefficients of 1D gaussian kernel with sigma = 1
    //apply x-Kernal from column 1 to column cols - 2; //last but one column
    double coeffs[] = { 0.2442, 0.4026, 0.2442 };
    for (int row = 0; row < outFrame.rows; row++)
    {
        for (int col = 1; col < outFrame.cols - 1; col++)
        {
            outFrame.data[row* outFrame.cols + col] = inFrame.data[row* inFrame.cols + col - 1] * coeffs[0] +
                inFrame.data[row* inFrame.cols + col] * coeffs[1] +
                inFrame.data[row* inFrame.cols + col + 1] * coeffs[2];
        }
    }

}
```

Figure 10. *Serial code block for Gaussian filter*



```
class GaussianTBBX
{
    Mat m_inFrame;
    Mat m_outFrame;
    public:
        void operator()(const blocked_range<int>& r) const {
            double coeffs[] = { 0.2442, 0.4026, 0.2442 };
            for (size_t row = r.begin(); row != r.end(); ++row)
            {
                for (int col = 1; col < m_outFrame.cols - 1; col++)
                {
                    m_outFrame.data[row* m_outFrame.cols + col] = m_inFrame.data[row* m_inFrame.cols + col - 1]
                        * coeffs[0] + m_inFrame.data[row* m_inFrame.cols + col] * coeffs[1] +
                        m_inFrame.data[row* m_inFrame.cols + col + 1] * coeffs[2];
                }
            }
        }
        GaussianTBBX(Mat inFrame, Mat outFrame) :
            m_inFrame(inFrame), m_outFrame(outFrame)
        {}
};

void CannyTBB::GaussianFilter(Mat& inFrame, Mat& outFrame)
{
    parallel_for(tbb::blocked_range<int>(0, outFrame.rows - 1),
        GaussianTBBX(inFrame, outFrame));
}
```

Figure 11. *Parallel code block for Gaussian filter*

Sample original image frame extracted from video is shown in figure 12. Output image after Gaussian filtering is as shown in figure 13.

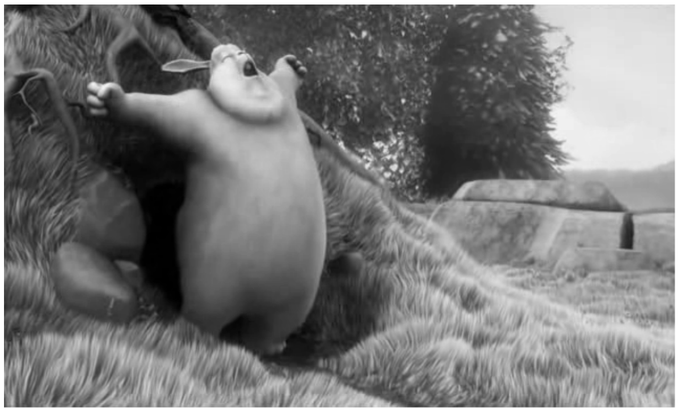


Figure 12. *Original Image*

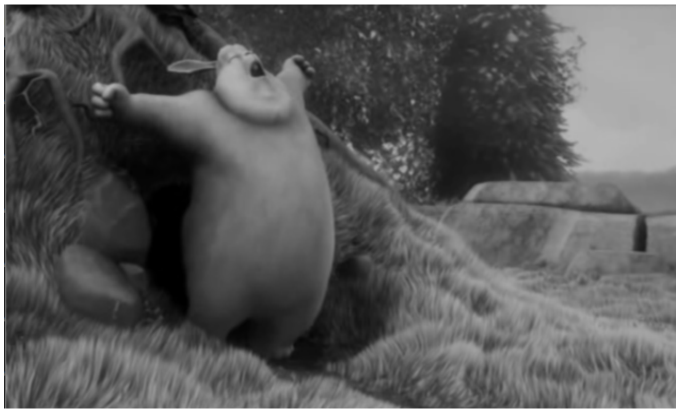


Figure 13. *Image after Gaussian filtering*

4.2. Sobel operator to find intensity gradient of image

After smoothing the image and eliminating the noise, the next step is to find the edge strength by taking the gradient of the image. The Sobel operator performs a 2-D spatial gradient measurement on an image. It uses a pair of 3x3 convolution masks, one estimating the gradient in the x-direction (columns) and the other estimating the gradient in the y-direction (rows).

$$g_x = \begin{array}{|c|c|c|}\hline -1 & 0 & 1 \\ \hline -2 & 0 & 2 \\ \hline -1 & 0 & 1 \\ \hline \end{array} \qquad g_y = \begin{array}{|c|c|c|}\hline -1 & -2 & -1 \\ \hline 0 & 0 & 0 \\ \hline 1 & 2 & 1 \\ \hline \end{array}$$

Figure 14. *Sobel Operator Gx and Gy [14]*

Sobel Gx and Gy masks shown in figure 14, estimates gradient x direction and y direction respectively.The image is convolved with both kernels to approximate the derivatives in horizontal and vertical change. At each given point, magnitude of the gradient can be approximated with:

Magnitude:
$$g = \sqrt{g_x^2 + g_y^2}$$

Orientation:
$$\Theta = \tan^{-1}\left(\frac{g_y}{g_x}\right)$$

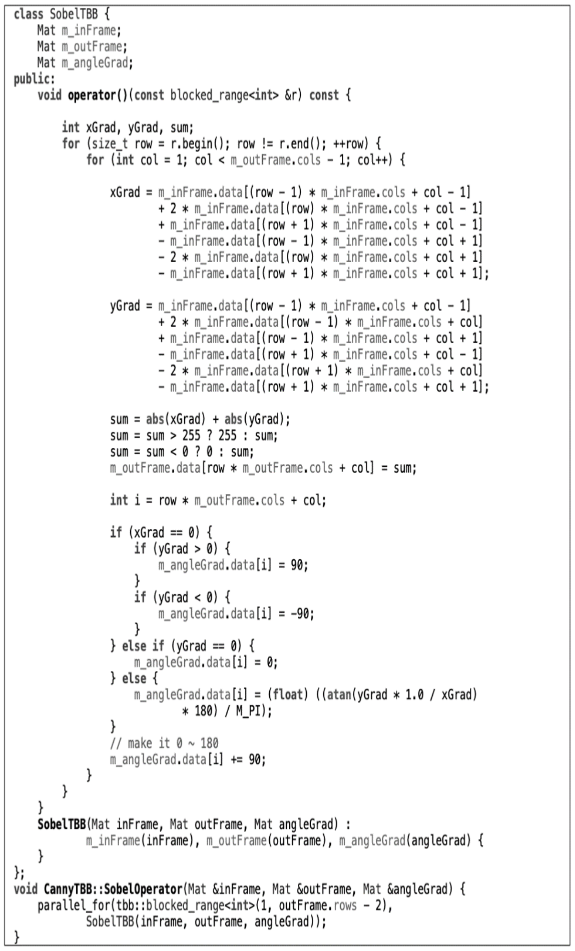
Figure 15. *The Magnitude and Orientation [14]*

```
//Apply Sobel filter to find x and y gradients
void CannySeq::SobelOperator(Mat &inFrame, Mat &outFrame, Mat& angleGrad) {

    CopyEdgePixelData(inFrame, outFrame);

    int xGrad, yGrad, sum;
    //calculate x gradient
    for (int row = 1; row < outFrame.rows - 1; row++) {
        for (int col = 1; col < outFrame.cols - 1; col++) {

            xGrad = inFrame.data[(row - 1) * inFrame.cols + col - 1]
                    + 2 * inFrame.data[(row) * inFrame.cols + col - 1]
                    + inFrame.data[(row + 1) * inFrame.cols + col - 1]
                    - inFrame.data[(row - 1) * inFrame.cols + col + 1]
                    - 2 * inFrame.data[(row) * inFrame.cols + col + 1]
                    - inFrame.data[(row + 1) * inFrame.cols + col + 1];

            yGrad = inFrame.data[(row - 1) * inFrame.cols + col - 1]
                    + 2 * inFrame.data[(row - 1) * inFrame.cols + col]
                    + inFrame.data[(row - 1) * inFrame.cols + col + 1]
                    - inFrame.data[(row + 1) * inFrame.cols + col - 1]
                    - 2 * inFrame.data[(row + 1) * inFrame.cols + col]
                    - inFrame.data[(row + 1) * inFrame.cols + col + 1];

            sum = abs(xGrad) + abs(yGrad);
            sum = sum > 255 ? 255 : sum;
            sum = sum < 0 ? 0 : sum;
            outFrame.data[row * outFrame.cols + col] = sum;

            int i = row * outFrame.cols + col;

            if (xGrad == 0) {
                if (yGrad > 0) {
                    angleGrad.data[i] = 90;
                }
                if (yGrad < 0) {
                    angleGrad.data[i] = -90;
                }
            } else if (yGrad == 0) {
                angleGrad.data[i] = 0;
            } else {
                angleGrad.data[i] = (float) ((atan(yGrad * 1.0 / xGrad) * 180)
                        / M_PI);
            }
            // make it 0 ~ 180
            angleGrad.data[i] += 90;
        }
    }
}
```

Figure 16. *Sobel Operator serial code*

Due to Sobel operator's smoothing effect (Gaussian smoothing), it is less sensitive to noise present in images. On the other hand, smoothing affects the accuracy of edge detection. In other words, the Sobel method does not produce image with high accuracy for edge detection, but its quality is adequate enough to be used in numerous applications.

The primary advantages of the Sobel operator lie in its simplicity. The Sobel method provides a approximation to the gradient magnitude. Another advantage of the Sobel operator is it can detect edges and their orientations. In this cross operator, the detection of edges and their orientations is said to be simple due to the approximation of the gradient



```
class SobelTBB {
    Mat m_inFrame;
    Mat m_outFrame;
    Mat m_angleGrad;
public:
    void operator()(const blocked_range<int> &r) const {

        int xGrad, yGrad, sum;
        for (size_t row = r.begin(); row != r.end(); ++row) {
            for (int col = 1; col < m_outFrame.cols - 1; col++) {

                xGrad = m_inFrame.data[(row - 1) * m_inFrame.cols + col - 1]
                        + 2 * m_inFrame.data[(row) * m_inFrame.cols + col - 1]
                        + m_inFrame.data[(row + 1) * m_inFrame.cols + col - 1]
                        - m_inFrame.data[(row - 1) * m_inFrame.cols + col + 1]
                        - 2 * m_inFrame.data[(row) * m_inFrame.cols + col + 1]
                        - m_inFrame.data[(row + 1) * m_inFrame.cols + col + 1];

                yGrad = m_inFrame.data[(row - 1) * m_inFrame.cols + col - 1]
                        + 2 * m_inFrame.data[(row - 1) * m_inFrame.cols + col]
                        + m_inFrame.data[(row - 1) * m_inFrame.cols + col + 1]
                        - m_inFrame.data[(row + 1) * m_inFrame.cols + col - 1]
                        - 2 * m_inFrame.data[(row + 1) * m_inFrame.cols + col]
                        - m_inFrame.data[(row + 1) * m_inFrame.cols + col + 1];

                sum = abs(xGrad) + abs(yGrad);
                sum = sum > 255 ? 255 : sum;
                sum = sum < 0 ? 0 : sum;
                m_outFrame.data[row * m_outFrame.cols + col] = sum;

                int i = row * m_outFrame.cols + col;

                if (xGrad == 0) {
                    if (yGrad > 0) {
                        m_angleGrad.data[i] = 90;
                    }
                    if (yGrad < 0) {
                        m_angleGrad.data[i] = -90;
                    }
                } else if (yGrad == 0) {
                    m_angleGrad.data[i] = 0;
                } else {
                    m_angleGrad.data[i] = (float) ((atan(yGrad * 1.0 / xGrad)
                            * 180) / M_PI);
                }
                // make it 0 ~ 180
                m_angleGrad.data[i] += 90;
            }
        }
    }
    SobelTBB(Mat inFrame, Mat outFrame, Mat angleGrad) :
            m_inFrame(inFrame), m_outFrame(outFrame), m_angleGrad(angleGrad) {
    }
};
void CannyTBB::SobelOperator(Mat &inFrame, Mat &outFrame, Mat &angleGrad) {
    parallel_for(tbb::blocked_range<int>(1, outFrame.rows - 2),
            SobelTBB(inFrame, outFrame, angleGrad));
}
```

Figure 17. *Sobel Operator parallel code*

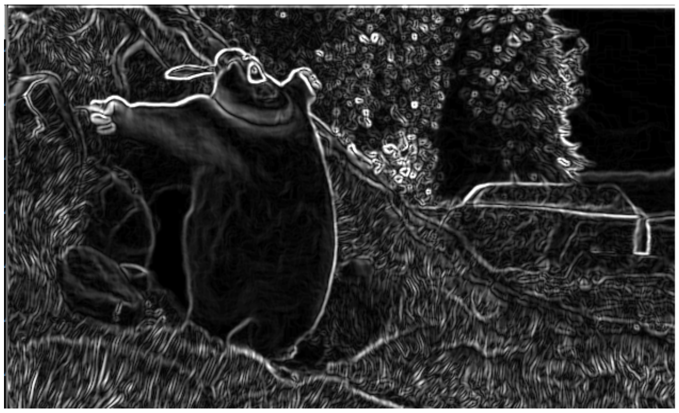


Figure 18. *Output Image after Sobel operator*

magnitude. On the other hand, there exist some disadvantages of the Sobel method. That is, it is sensitive to the noise. The magnitude of the edges will degrade as the level of noise present in image increases. As a result, Sobel operator accuracy suffers as the magnitude of the edges decreases. Overall, the Sobel method cannot produce accurate edge detection with thin and smooth edge.

Figure 16 and 17 shows the serial code of Sobel operator and it's parallel version respectively.
Output image after applying Sobel operator is as shown in figure 18.

4.3. Non-maximum Suppression

Ideally, the final image should have thin edges. Non-maximum suppression is performed to thin out the edges. After the edge directions are known, non-maximum suppression is applied.

Non-maximum suppression is used to trace along the gradient in the edge direction and compare the value perpendicular to the gradient. Two perpendicular pixel values are compared with the value in the edge direction. If their value is lower than the pixel on the edge then they are suppressed i.e. their pixel value is changed to 0, else the higher pixel value is set as the edge and the other two suppressed with a pixel value of 0.

The algorithm goes through all the points on the gradient intensity matrix and finds the pixels with the maximum value in the edge directions. The upper left corner red box present in the figure 19, represents an intensity pixel of the Gradient Intensity matrix being processed. The corresponding edge direction is represented by the arrow with an angle of -pi radians (+/-180 degrees) [15]. In the figure 20,

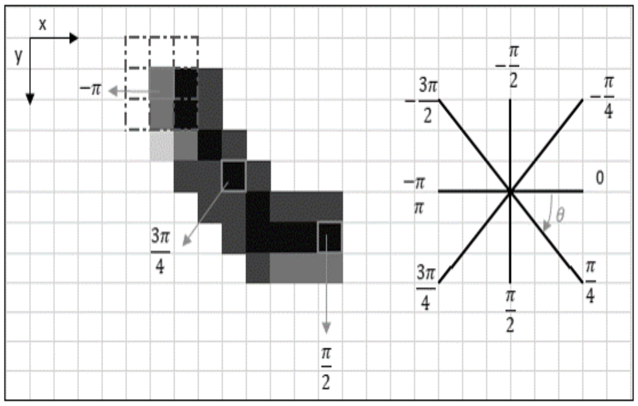


Figure 19. *Non-maximum suppression example [15]*

the edge direction is the dotted line (horizontal from left to right). The purpose of the algorithm is to check if the pixels on the same direction are more or less intense than the ones being processed. In the figure 20, the pixel (i, j) is being processed, and the pixels on the same direction are

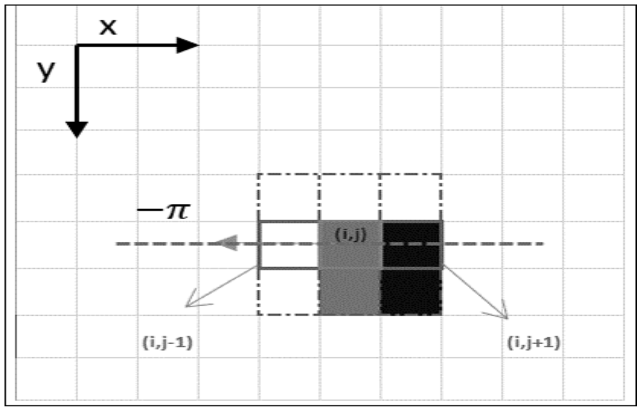
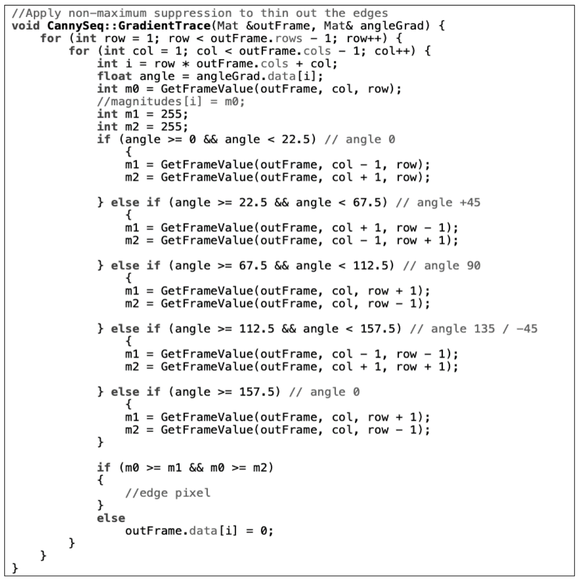


Figure 20. *Non-maximum suppression example [15]*

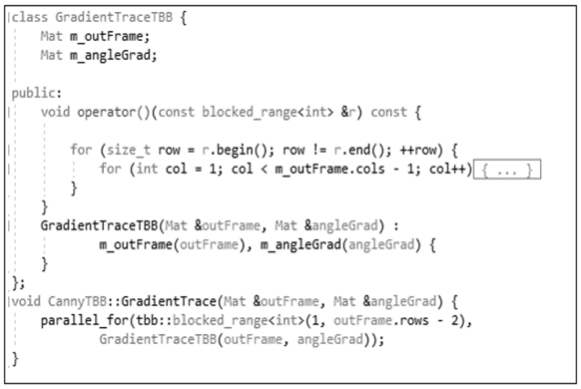
highlighted in blue (i, j-1) and (i, j+1). If one those two pixels are more intense than the one being processed, then only the more intense one is kept. Pixel (i, j-1) seems to be more intense, because it is white (value of 255). Hence, the intensity value of the current pixel (i, j) is set to 0. If there are no pixels in the edge direction having more intense values, then the value of the current pixel is kept [15].
Serial code for non-maximum suppression is as shown in figure 21



```
//Apply non-maximum suppression to thin out the edges
void CannySeq::GradientTrace(Mat &outFrame, Mat& angleGrad) {
    for (int row = 1; row < outFrame.rows - 1; row++) {
        for (int col = 1; col < outFrame.cols - 1; col++) {
            int i = row * outFrame.cols + col;
            float angle = angleGrad.data[i];
            int m0 = GetFrameValue(outFrame, col, row);
            //magnitudes[i] = m0;
            int m1 = 255;
            int m2 = 255;
            if (angle >= 0 && angle < 22.5) // angle 0
                {
                m1 = GetFrameValue(outFrame, col - 1, row);
                m2 = GetFrameValue(outFrame, col + 1, row);

            } else if (angle >= 22.5 && angle < 67.5) // angle +45
                {
                m1 = GetFrameValue(outFrame, col + 1, row - 1);
                m2 = GetFrameValue(outFrame, col - 1, row + 1);

            } else if (angle >= 67.5 && angle < 112.5) // angle 90
                {
                m1 = GetFrameValue(outFrame, col, row + 1);
                m2 = GetFrameValue(outFrame, col, row - 1);

            } else if (angle >= 112.5 && angle < 157.5) // angle 135 / -45
                {
                m1 = GetFrameValue(outFrame, col - 1, row - 1);
                m2 = GetFrameValue(outFrame, col + 1, row + 1);

            } else if (angle >= 157.5) // angle 0
                {
                m1 = GetFrameValue(outFrame, col, row + 1);
                m2 = GetFrameValue(outFrame, col, row - 1);
            }

            if (m0 >= m1 && m0 >= m2)
            {
                //edge pixel
            }
            else
                outFrame.data[i] = 0;
        }
    }
}
```

Figure 21. *Serial code block for Non-maximum suppression*

Parallel_for for non-maximum suppression is as shown in figure 22
Output image after applying non-maximum suppression is as shown in figure 23.



```
class GradientTraceTBB {
    Mat m_outFrame;
    Mat m_angleGrad;

public:
    void operator()(const blocked_range<int> &r) const {

        for (size_t row = r.begin(); row != r.end(); ++row) {
            for (int col = 1; col < m_outFrame.cols - 1; col++){ ... }
        }
    }
    GradientTraceTBB(Mat &outFrame, Mat &angleGrad) :
            m_outFrame(outFrame), m_angleGrad(angleGrad) {
    }
};
void CannyTBB::GradientTrace(Mat &outFrame, Mat &angleGrad) {
    parallel_for(tbb::blocked_range<int>(1, outFrame.rows - 2),
            GradientTraceTBB(outFrame, angleGrad));
}
```

Figure 22. *Parallel_for for Non-maximum suppression*

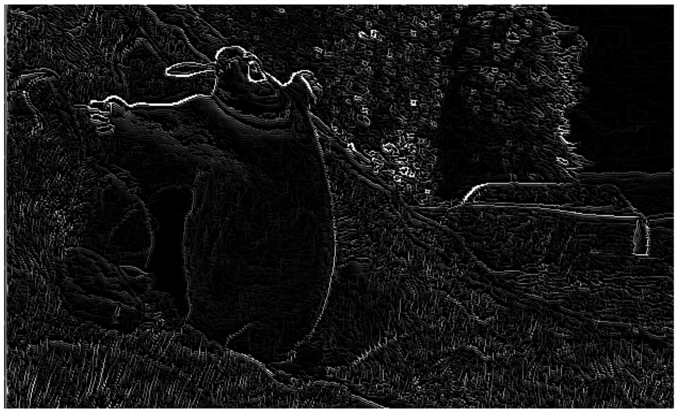


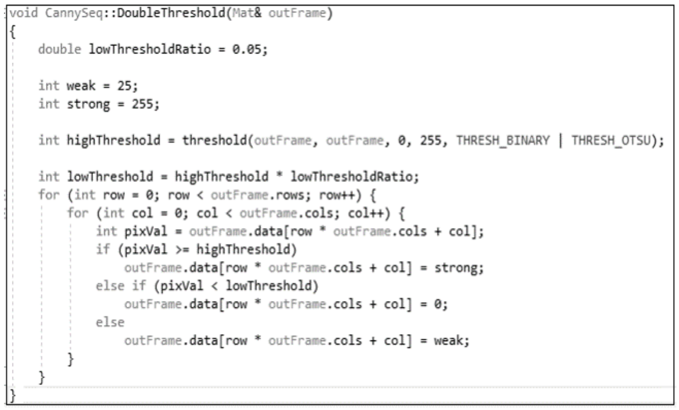
Figure 23. *Output Image after non-maximum suppression*

4.4. Double-threshold detection

After application of non-maximum suppression, remaining edge pixels provide a more accurate representation of real edges in an image.

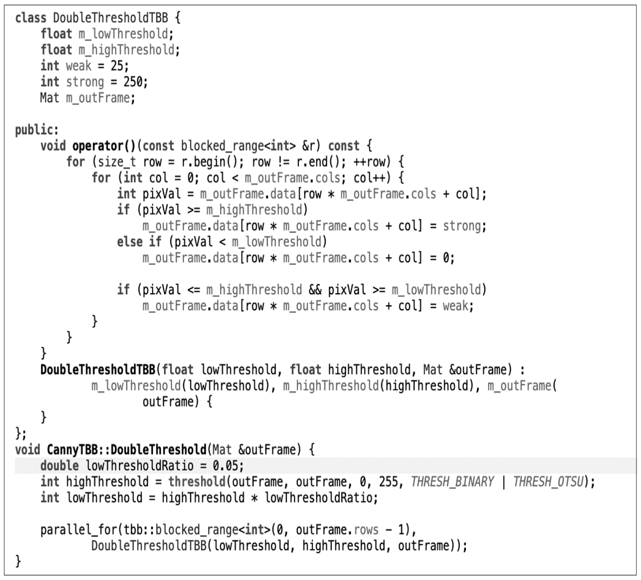
However, some edge pixels remain that are caused by noise and color variation. In order to account for these spurious responses, it is essential to filter out edge pixels with a weak gradient value and preserve edge pixels with a high gradient value. This is accomplished by selecting high and low threshold values. If an edge pixel's gradient value is higher than the high threshold value, it is marked as a strong edge pixel. If an edge pixel's gradient value is smaller than the high threshold value and larger than the low threshold value, it is marked as a weak edge pixel. If an edge pixel's value is smaller than the low threshold value, it will be suppressed. The two threshold values are empirically determined and their definition will depend on the content of a given input image. [4]

Output image after applying double-threshold is as shown in figure 26.



```
void CannySeq::DoubleThreshold(Mat& outFrame)
{
    double lowThresholdRatio = 0.05;

    int weak = 25;
    int strong = 255;

    int highThreshold = threshold(outFrame, outFrame, 0, 255, THRESH_BINARY | THRESH_OTSU);

    int lowThreshold = highThreshold * lowThresholdRatio;
    for (int row = 0; row < outFrame.rows; row++) {
        for (int col = 0; col < outFrame.cols; col++) {
            int pixVal = outFrame.data[row * outFrame.cols + col];
            if (pixVal >= highThreshold)
                outFrame.data[row * outFrame.cols + col] = strong;
            else if (pixVal < lowThreshold)
                outFrame.data[row * outFrame.cols + col] = 0;
            else
                outFrame.data[row * outFrame.cols + col] = weak;
        }
    }
}
```

Figure 24. *Serial code for double-threshold*



```
class DoubleThresholdTBB {
    float m_lowThreshold;
    float m_highThreshold;
    int weak = 25;
    int strong = 250;
    Mat m_outFrame;

public:
    void operator()(const blocked_range<int> &r) const {
        for (size_t row = r.begin(); row != r.end(); ++row) {
            for (int col = 0; col < m_outFrame.cols; col++) {
                int pixVal = m_outFrame.data[row * m_outFrame.cols + col];
                if (pixVal >= m_highThreshold)
                    m_outFrame.data[row * m_outFrame.cols + col] = strong;
                else if (pixVal < m_lowThreshold)
                    m_outFrame.data[row * m_outFrame.cols + col] = 0;

                if (pixVal <= m_highThreshold && pixVal >= m_lowThreshold)
                    m_outFrame.data[row * m_outFrame.cols + col] = weak;
            }
        }
    }
    DoubleThresholdTBB(float lowThreshold, float highThreshold, Mat &outFrame) :
            m_lowThreshold(lowThreshold), m_highThreshold(highThreshold), m_outFrame(
                outFrame) {
    }
};
void CannyTBB::DoubleThreshold(Mat &outFrame) {
    double lowThresholdRatio = 0.05;
    int highThreshold = threshold(outFrame, outFrame, 0, 255, THRESH_BINARY | THRESH_OTSU);
    int lowThreshold = highThreshold * lowThresholdRatio;

    parallel_for(tbb::blocked_range<int>(0, outFrame.rows - 1),
            DoubleThresholdTBB(lowThreshold, highThreshold, outFrame));
}
```

Figure 25. *Parallel code block for Double Threshold suppression*

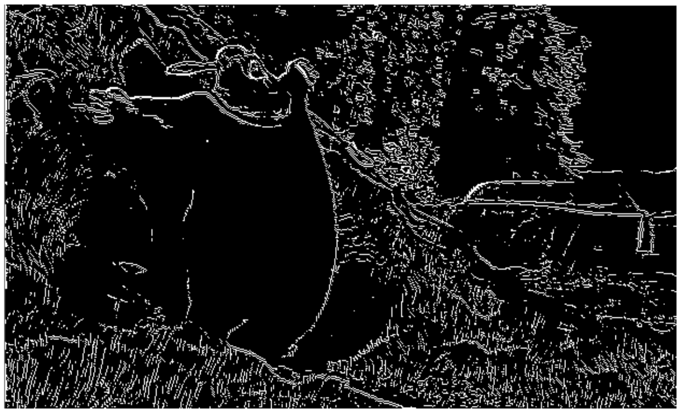
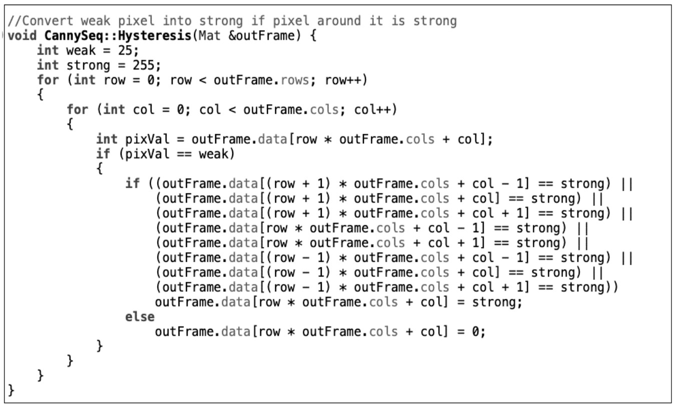


Figure 26. *Output Image after double-threshold*

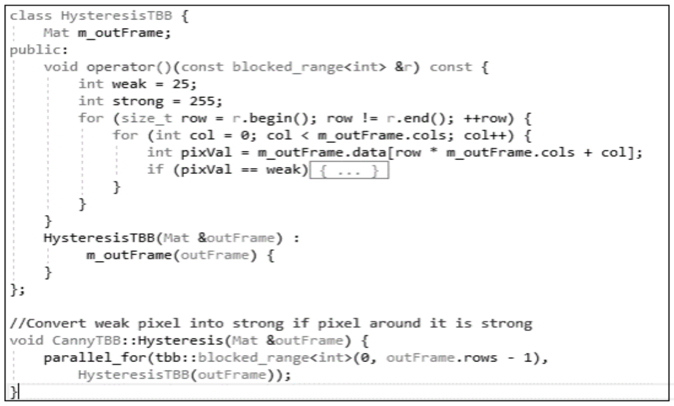
4.5. Edge tracking by hysteresis

Finally, hysteresis is used as a means of eliminating streaking. Streaking is the breaking up of an edge contour caused by the operator output fluctuating above and below the threshold. If a single threshold, T1 is applied to an image, and an edge has an average strength equal to T1, then due to noise, there will be instances where the edge dips below the threshold. Equally it will also extend above the threshold making an edge look like a dashed line. To avoid this, hysteresis uses 2 thresholds, a high and a low. Any pixel in the image that has a value greater than T1 is presumed to be an edge pixel, and is marked as such immediately. Then, any pixels that are connected to this edge pixel and that have a value greater than T2 are also selected as edge pixels. If you think of following an edge, you need a gradient of T2 to start but you don't stop till you hit a gradient below T1.



```
//Convert weak pixel into strong if pixel around it is strong
void CannySeq::Hysteresis(Mat &outFrame) {
    int weak = 25;
    int strong = 255;
    for (int row = 0; row < outFrame.rows; row++)
    {
        for (int col = 0; col < outFrame.cols; col++)
        {
            int pixVal = outFrame.data[row * outFrame.cols + col];
            if (pixVal == weak)
            {
                if ((outFrame.data[(row + 1) * outFrame.cols + col - 1] == strong) ||
                    (outFrame.data[(row + 1) * outFrame.cols + col] == strong) ||
                    (outFrame.data[(row + 1) * outFrame.cols + col + 1] == strong) ||
                    (outFrame.data[row * outFrame.cols + col - 1] == strong) ||
                    (outFrame.data[row * outFrame.cols + col + 1] == strong) ||
                    (outFrame.data[(row - 1) * outFrame.cols + col - 1] == strong) ||
                    (outFrame.data[(row - 1) * outFrame.cols + col] == strong) ||
                    (outFrame.data[(row - 1) * outFrame.cols + col + 1] == strong))
                    outFrame.data[row * outFrame.cols + col] = strong;
                else
                    outFrame.data[row * outFrame.cols + col] = 0;
            }
        }
    }
}
```

Figure 27. *Serial code for hystersis*



```
class HysteresisTBB {
    Mat m_outFrame;
public:
    void operator()(const blocked_range<int> &r) const {
        int weak = 25;
        int strong = 255;
        for (size_t row = r.begin(); row != r.end(); ++row) {
            for (int col = 0; col < m_outFrame.cols; col++) {
                int pixVal = m_outFrame.data[row * m_outFrame.cols + col];
                if (pixVal == weak){ ... }
            }
        }
    }
    HysteresisTBB(Mat &outFrame) :
         m_outFrame(outFrame) {
    }
};

//Convert weak pixel into strong if pixel around it is strong
void CannyTBB::Hysteresis(Mat &outFrame) {
    parallel_for(tbb::blocked_range<int>(0, outFrame.rows - 1),
        HysteresisTBB(outFrame));
}
```

Figure 28. *Parallel_for code for hystersis*

Output image after applying hysteresis is as shown in figure 29.

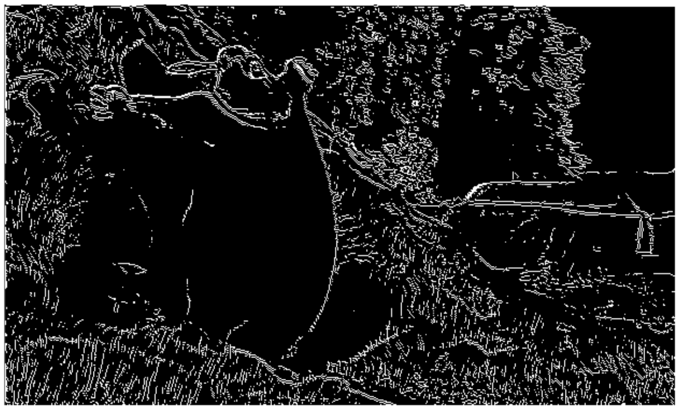


Figure 29. *Output Image after hysteresis*

5. Experimental Results

To measure the speed up achieved by using Intel TBB over sequential approach, we designed an command line application in c++ which implements the Canny Edge detector. It uses two modes, "seq" for running sequential and "tbb" for running Intel TBB implementation. It also takes video path as input. (Command line for application : EdgeDetectionTBB -mode <seq/tbb> -inp <video path>) Application extracts image frames form video and calculate the total time for edge detection on all frames. Total time calculated is divided by number of frames thus calculating the average time per frame for edge detection. Application measures the speedup achieved by using tbb parallel_for loop over sequential for loop. Application is run on three different image resolutions to check the speedup relation with image size. Three different resolutions used are 1280x720, 640x360 and 360x240. We measured the speedup on 4 core system and 6 core system.
Experiments are performed on following two system configurations:
System Configuration 1

CPU	Intel Core i5 @ 2.4GHz
RAM	8 GB
OS	Windows 10
No. of processors	4

System Configuration 2

CPU	Intel Core i7 @ 2.6GHz
RAM	16 GB
OS	Windows 10
No. of processors	6

Below are the results observed for Canny Edge detection on 4 core system.
Simulation Results

Image Size	Sequential	TBB
1280x720	(69554, 71399)	(35167, 36286)
640x360	(17947, 18127)	(9180, 9214)
360x240	(5010, 5128)	(2893, 3173)

For resolution 1280x720 and 640x360, speed up of approximately 1.9x is observed. For resolution 360x240, speed up of approximately 1.6x is observed. Approximately double speedup is observed when data size is large. For shorter loops or data size, sppedup is less. This may be because of the overhead to wake up threads from the pool of pre-allocated threads and dispatch logical tasks to threads.

Below are the results observed for Canny Edge detection on 6 core system.
Simulation Results

Image Size	Sequential	TBB
1280x720	(55416, 56975)	(16626, 17622)
640x360	(13383, 13919)	(4663, 4828)
360x240	(3665, 4041)	(1737, 1879)

For 1280x720 resolution, speedup is approximately 3.2x w.r.t. sequential. For 640x360 resolution, speedup is approximately 2.8x w.r.t. sequential. For 360x240 resolution, speedup is approximately 2.1x w.r.t. sequential. In TBB mode, 6 core system has almost double speedup as compared to 4 core system.

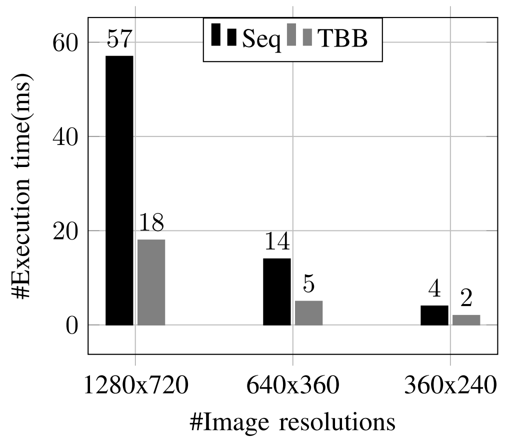


Figure 30. Canny processing time for Sequential and TBB (ms) on 6 core system

6. Conclusion and Future Scope

Intel TBB platform can be used to increase the processing speed of applications using data parallelism. More processing speed can be observed with larger data size. Scalability of application can be improved with increasing number of cores in the system i.e. more the number of cores, faster the processing speed using TBB.

Our application measures the speedup achieved by using tbb parallel_for loop over sequential for loop. We used

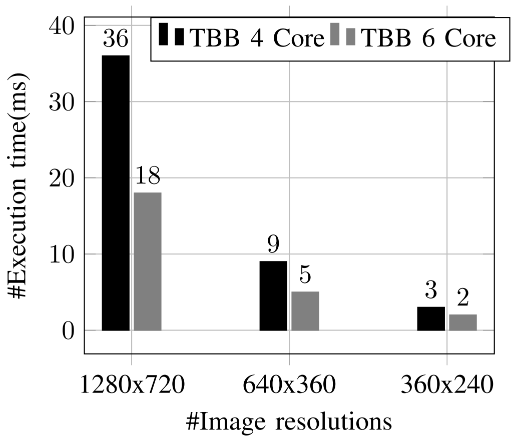


Figure 31. TBB Processing time (ms) comparison for 6 core and 4 core

blocked_range function for parallel_for which breaks the iteration space into 1-D chunks and applies parallel operations on it. If we had more time to research about blocked_range2d function which breaks the iteration space into 2-D chunks, speedup could be noted for this function. Intel TBB suggests to use blocked_range2d for nested for loops so that more performance gain can be achieved. Further in practical use, parallel_pipeline can be used to parallelize the operations on each frame so that calculations on individual frames can run in parallel. It will be good research subject to verify speedup achieved by such function for huge data and large number of calculations.

References

[1] Sushil Kumar Sah , Dinesh Naik, "Parallelizing Doolittle Algorithm Using TBB" 2014 International Conference on Parallel, Distributed and Grid Computing. [Online].Available: `https://ieeexplore.ieee.org/document/7030707`

[2] Cheong Ghil Kim, "Accelerating Multimedia Applications using Intel Threading Building Blocks on Multi-core Processors", 2011 International Conference on Information Science and Applications [Online]. Available: `https://ieeexplore.ieee.org/document/5772423`

[3] Intel Threading Building Blocks tutorials. [Online]. Available: `https://software.intel.com/en-us/tbb-tutorial`

[4] Canny Edge Detector [Online]. Available: `https://en.wikipedia.org/wiki/Canny_edge_detector`

[5] Edge detection, from Wikipedia: `https://en.wikipedia.org/wiki/Edge_detection`

[6] Parallator Transformation towards C++ parallelism using TBB. [Online]. Available: `https://eprints.hsr.ch/285/1/parallator.pdf`

[7] Visual Studio 2017 download `https://docs.microsoft.com/en-us/visualstudio/windows/?view=vs-2017`

[8] Opencv 4.1.2 download `https://github.com/opencv/opencv/releases`

[9] Intel TBB download `https://software.seek.intel.com/performance-libraries`

[10] Intel TBB Document 1: `https://www.inf.ed.ac.uk/teaching/courses/ppls/TBBtutorial.pdf`

[11] Intel TBB Document 2: `http://www.cs.cmu.edu/afs/cs/academic/class/15499-s09/www/handouts/TBB-HPCC07.pdf`

[12] Intel TBB Document 3: `https://www.nersc.gov/assets/Uploads/Intel-TBB-Mar2017.PDF`

[13] Gaussian Filtering `https://www.cs.auckland.ac.nz/courses/compsci373s1c/PatricesLectures/Gaussian%20Filtering_1up.pdf`

[14] Sobel Edge Detector `https://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm`

[15] Algorithm Code Reference `https://towardsdatascience.com/canny-edge-detection-step-by-step-in-python-computer-vision-b49c3a2d8123`