A significant contribution of their work is the ability to use an implicit smoothing technique to enable the algorithm to work on noisy images. This was particularly important as the Prewitt edge detection algorithm is known to be sensitive to noise and therefore may not generate satisfactory results in the presence of noise. The implicit smoothing technique used in [6] calculates the gradient magnitude of the eight directions as the basis to find the final magnitude to reduce the effect of noise. Our experiments show that the gradient based implicit smoothing approach will not be very effective when the image is impacted with high noise percentage. In our work, we propose to use an explicit smoothing technique that can be applied to noisy images as a prefix step to the Prewitt edge detection algorithm [9]. Our results show an improvement in the quality and overall execution time even when we include the prefix smoothing step. In this paper we present an efficient parallel eight directions Prewitt edge detection augmented with explicit smoothing and iterative thresholding functionality capable of producing fast and accurate results.

The Pseudocode of the eight directions edge detection is very similar to the bidirectional presented in Figure 2 with additional 6 windows to convolve the image with a total of eight windows proposed in [6].

```
/* Where f(x,y) is the input image, k1(x,y)and k2(x,y)are the two mask windows in
direction 0" and 90" respectively and g(x,y) is the output image */
I[imgHeight, imgWidth] := imRead(image);
                    /* imgHeight: image's height, imgWidth: image's width. */
k1[maskHeight,mWidth] := imRead(Mask);
                    /* maskHeight: mask's height, maskWidth: mask's width. */
k2[maskHeight,mWidth] := imRead(Mask);
                    /* maskHeight: mask's height, maskWidth: mask's width. */
h := (maskHeight-1)/2 ;
w := (maskWidth-1)/2 ;
for y := 0 until imgHeight do
   for x := 0 until imgWidth do
     { sum = 0 ; sum1 = 0 ; sum2 = 0 ; /* initialization */
   for i := -h until h do
      for j := -w until w do
        { sum1 + = k1(j,i)*I(x-j,y-i) ; /*convolution in the first direct*/
          sum2 + = k2(j,i)*I(x-j,y-i); /* convolution in the second direct*/
          sum = max(sum1, sum2) } /* select the max intensity */
      g(x,y) = sum;   }          /* result image */
```

Figure 2: Original two directions Prewitt edge detection's Pseudocode.

Section 2 describes the design of the augmented parallel eight directions Prewitt algorithm. Section 3 covers the experimental results and analysis. Section 4 summarizes the paper and conclusions.

2 Augmented parallel eight direction Prewitt edge detection algorithm

The original bidirectional Prewitt algorithm is known for its good computational complexity but noise sensitivity; lack of an affective smoothing mechanism has mostly disqualified its usage. Analytical studies conducted in [7] indicate that Prewitt edge detection algorithm cannot be used along with practical images that are often corrupted with impulse noise as well as Gaussian and Poisson noises. Majority of images are often inflicted by varying degree of noise caused by transmission channels and camera sensors [7]. Furthermore, Prewitt algorithm does not incorporate a thresholding mechanism in order to produce binary images. Most object detection applications rely on background subtraction. The presence of only two values in the resulting binary image, one representing the object and the other representing the background, is desirable in object detection applications [10]. We have augmented our parallel algorithm with both smoothing and thresholding mechanisms. In this section we will first discuss our choices for smoothing and thresholding strategies and then present the description of complete parallel application.

2.1 Selecting an appropriate filter to use as prefix step to eight direction Prewitt

The stand-alone Prewitt algorithm does not incorporate any mechanisms to deal with noisy images. The Canny edge detection algorithm uses Gaussian filter or Blurring as a prefix smoothing mechanism [3, 5] to address the noise issue. Blurring can destroy some real edges during the process of noise suppression. Bilateral is known to be a better choice because it has less negative impact on real edges than Blurring. To select a suitable smoothing method, we conducted several experiments using Blurring, Median Blurring, Gaussian, Bilateral, Median filtering and an improved version of Median filter. Our improved Median filter, added as a prior step to localization in the eight direction Prewitt algorithm, shows the best results for suppressing impulse noises such as Salt and Pepper and at the same time is able to suppress other types of noises such as Gaussian and Poisson. Median filter is a nonlinear filter used to remove impulse noise with the least negative impact on the real edges [5, 11, 12]. This filter starts arranging all the pixels within the range of the specified window in ascending order to choose the median value. This approach helps keep some real values unchanged as opposed to the blurring technique that takes the average of all pixels within that neighborhood. The main idea behind our improved Median filter is to allow the window size be variable and not fixed as in the standard Median for the sake of not only better noise suppression capability but also operations reduction [12]. Larger window sizes can be used effectively in processing images with higher noise percentage using the same mechanism of Median filter. In the current version, we provide the users full control to judge the intensity of the noise in order to determine the desired window size. We recommend to start with a window of size 3x3, considered to be the standard case (level 1). If the resulting output reflects detection of false edges, users can increase the size of the window to 5x5(level 2), 7x7(level 3), 9x9(level 4), or double 9x9 (level 5) respectively. Double 9x9 stands for re-smoothing the results of level 4 with 9x9 window size. Our results demonstrate clear detection of edges in the presence of noise when smoothing is enabled. The experimental results are presented in the results section, Table 1 and Figure 6.

2.2 Dynamic Iterative Thresholding

Thresholding is desirable in many applications; we have added this functionality to the algorithm to be used when needed. There are many thresholding techniques available ranging from visual judgment using trial and error to reliance on global or local methods [5, 13, 14]. In this work we have selected to add the basic global thresholding method due to its computational simplicity, acceptable quality and applicability to a variety of applications [5]. The technique works iteratively to find the thresholding value as described in Figure 3.

Step 1: Start with initial guess for the thresholding value T.
/* For faster convergence, choose initial T to be the average of all intensities of the assigned work [5] */
Step 2: For each pixel, marked as group 1 (G_1) or group 2 (G_2):
a. if img[i,j] > T, img[i,j] ε G_1
b. else img[i,j] ε G_2
Step 3: For each group, find the average of intensities Av_1 and Av_2 respectively.
Step 4: Find the new threshold value: $T_{new} = (Av_1 + Av_2)/2$.
Step 5: Stop if $|T_{new} - T| \leq$ tolerant value; Otherwise $T = T_{new}$ and repeat the process from step 2.

Figure 3: The procedure of the Basic Global Thresholding.

2.3 Parallel application

Adding a smoothing mechanism with variable window sizes, six additional directions to the localization step and a global thresholding mechanism adds computational complexity to the original bidirectional Prewitt algorithm. These additional computations affect the overall performance. A parallel version of the proposed algorithm for the new shared memory MIMD multicore platforms is designed and implemented in order to speed up the computation. The parallel algorithm is designed and implemented using C/C++ as a base language and two open source libraries OpenMP and the open computer vision library (OpenCV) to overcome complexities added to the original algorithm. OpenCV library supports hundreds of optimized image processing algorithms mainly designed for real time applications contributing to the field of computer vision specifically [13, 15]. These libraries lend themselves well to parallel processing [13, 16]. One of the main challenges working on shared memory multiprocessors is the work distribution [1, 17]. Work, for good distribution, can be divided into rows, columns or blocks. Using appropriate mechanisms for data partitioning not only provides an independent chunk of data that can be processed concurrently but also will add flexibility in tuning the algorithm to run more efficiently on different architectural platforms to enhance data locality that in turn reduces the number of time-consuming cache misses. Work distribution starts by allowing each of the processors to copy its assigned private data to its local memory. To gain efficiency, it is desired to decrease the number of times needed to copy data from slower shared memory to local memory at each computational unit. The algorithm starts dividing the image into a number of equal sized tiles (sub-images). Parallel processes will work independently on different sub-images using a self-scheduling technique for work distribution. Each processor applies smoothing, localization, and iterative thresholding before writing the processed data to its final location as shown in Figure 4.

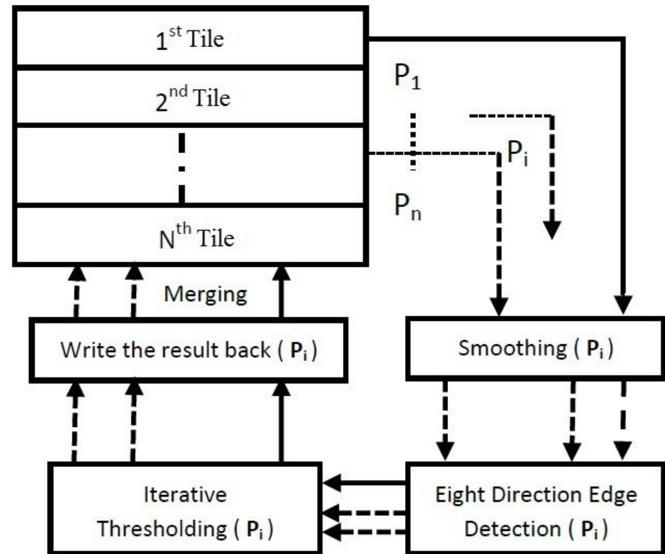

Figure 4: Processing life cycle.

As seen in Figure 4, the parallelism is applied at the highest possible level in which all work, in between the time of loading the main image up until merging sub-images, is done in parallel. We are presenting only partitioning into rows only. Yet our implementation supports all three partitioning methods (rows, columns, and blocks). The original input image is defined as an object (Region Of Interest, ROI) in the OpenCV library resulting in the need for some synchronization when the final result is written back to this object. This is a requirement enforced by OpenCV to guarantee data integrity, therefore, each processor will write the data to its original region of interest atomically. It is best to choose self-scheduling instead of pre-scheduling for good work balance. This will prevent the problem of having some processors idle while the others have excess work due to varying amount work required in each region of the image. It is important to note that sub-images are completely treated individually and are processed within the iterative thresholding mechanism separately. This strategy, according to our experiments, can result in better detection where more real edges are detected. Self-scheduling will overcome the difference in time required by applying the iterative thresholding mechanism on the assigned data as the number of computations may vary depending on the data itself.

A significant contribution of their work is the ability to use an implicit smoothing technique to enable the algorithm to work on noisy images. This was particularly important as the Prewitt edge detection algorithm is known to be sensitive to noise and therefore may not generate satisfactory results in the presence of noise. The implicit smoothing technique used in [6] calculates the gradient magnitude of the eight directions as the basis to find the final magnitude to reduce the effect of noise. Our experiments show that the gradient based implicit smoothing approach will not be very effective when the image is impacted with high noise percentage. In our work, we propose to use an explicit smoothing technique that can be applied to noisy images as a prefix step to the Prewitt edge detection algorithm [9]. Our results show an improvement in the quality and overall execution time even when we include the prefix smoothing step. In this paper we present an efficient parallel eight directions Prewitt edge detection augmented with explicit smoothing and iterative thresholding functionality capable of producing fast and accurate results.

The Pseudocode of the eight directions edge detection is very similar to the bidirectional presented in Figure 2 with additional 6 windows to convolve the image with a total of eight windows proposed in [6].

```
/* Where f(x,y) is the input image, k1(x,y)and  k2(x,y)are the two mask windows in
direction 0” and 90” respectively and  g(x,y) is the output image */
I[imgHeight, imgWidth]  := imRead(image);
                         /* imgHeight: image’s height, imgWidth: image’s width. */
k1[maskHeight,mWidth] := imRead(Mask);
                        /* maskHeight: mask’s height, maskWidth: mask’s width. */
k2[maskHeight,mWidth] := imRead(Mask);
                       /* maskHeight: mask’s height, maskWidth: mask’s width. */
h := (maskHeight-1)/2 ;
w := (maskWidth-1)/2 ;
for y := 0 until  imgHeight do
    for x := 0 until imgWidth do
      {  sum   = 0 ;  sum1 = 0 ;  sum2 = 0 ; /* initialization */
    for i := -h until h do
        for j := -w until w do
          { sum1 + = k1(j,i)*I(x-j,y-i) ; /*convolution in the first direct*/
            sum2 + = k2(j,i)*I(x-j,y-i); /* convolution in the second direct*/
             sum = max(sum1, sum2) } /* select the max intensity */
        g(x,y) = sum;   }                   /* result image */
```

Figure 2: Original two directions Prewitt edge detection’s Pseudocode.

Section 2 describes the design of the augmented parallel eight directions Prewitt algorithm. Section 3 covers the experimental results and analysis. Section 4 summarizes the paper and conclusions.

2 Augmented parallel eight direction Prewitt edge detection algorithm

The original bidirectional Prewitt algorithm is known for its good computational complexity but noise sensitivity; lack of an affective smoothing mechanism has mostly disqualified its usage. Analytical studies conducted in [7] indicate that Prewitt edge detection algorithm cannot be used along with practical images that are often corrupted with impulse noise as well as Gaussian and Poisson noises. Majority of images are often inflicted by varying degree of noise caused by transmission channels and camera sensors [7]. Furthermore, Prewitt algorithm does not incorporate a thresholding mechanism in order to produce binary images. Most object detection applications rely on background subtraction. The presence of only two values in the resulting binary image, one representing the object and the other representing the background, is desirable in object detection applications [10]. We have augmented our parallel algorithm with both smoothing and thresholding mechanisms. In this section we will first discuss our choices for smoothing and thresholding strategies and then present the description of complete parallel application.

2.1 Selecting an appropriate filter to use as prefix step to eight direction Prewitt

The stand-alone Prewitt algorithm does not incorporate any mechanisms to deal with noisy images. The Canny edge detection algorithm uses Gaussian filter or Blurring as a prefix smoothing mechanism [3, 5] to address the noise issue. Blurring can destroy some real edges during the process of noise suppression. Bilateral is known to be a better choice because it has less negative impact on real edges than Blurring. To select a suitable smoothing method, we conducted several experiments using Blurring, Median Blurring, Gaussian, Bilateral, Median filtering and an improved version of Median filter. Our improved Median filter, added as a prior step to localization in the eight direction Prewitt algorithm, shows the best results for suppressing impulse noises such as Salt and Pepper and at the same time is able to suppress other types of noises such as Gaussian and Poisson. Median filter is a nonlinear filter used to remove impulse noise with the least negative impact on the real edges [5, 11, 12]. This filter starts arranging all the pixels within the range of the specified window in ascending order to choose the median value. This approach helps keep some real values unchanged as opposed to the blurring technique that takes the average of all pixels within that neighborhood. The main idea behind our improved Median filter is to allow the window size be variable and not fixed as in the standard Median for the sake of not only better noise suppression capability but also operations reduction [12]. Larger window sizes can be used effectively in processing images with higher noise percentage using the same mechanism of Median filter. In the current version, we provide the users full control to judge the intensity of the noise in order to determine the desired window size. We recommend to start with a window of size 3x3, considered to be the standard case (level 1). If the resulting output reflects detection of false edges, users can increase the size of the window to 5x5(level 2), 7x7(level 3), 9x9(level 4), or double 9x9 (level 5) respectively. Double 9x9 stands for re-smoothing the results of level 4 with 9x9 window size. Our results demonstrate clear detection of edges in the presence of noise when smoothing is enabled. The experimental results are presented in the results section, Table 1 and Figure 6.

2.2 Dynamic Iterative Thresholding

Thresholding is desirable in many applications; we have added this functionality to the algorithm to be used when needed. There are many thresholding techniques available ranging from visual judgment using trial and error to reliance on global or local methods [5, 13, 14]. In this work we have selected to add the basic global thresholding method due to its computational simplicity, acceptable quality and applicability to a variety of applications [5]. The technique works iteratively to find the thresholding value as described in Figure 3.

Step 1: Start with initial guess for the thresholding value T.
/* For faster convergence, choose initial T to be the average of all intensities of the assigned work [5] */
Step 2: For each pixel, marked as group 1 (G_1) or group 2 (G_2):
a. if img[i,j] > T, img[i,j] ε G_1
b. else img[i,j] ε G_2
Step 3: For each group, find the average of intensities Av_1 and Av_2 respectively.
Step 4: Find the new threshold value: $T_{new} = (Av_1 + Av_2)/2$.
Step 5: Stop if $|T_{new} - T| \leq$ tolerant value; Otherwise $T = T_{new}$ and repeat the process from step 2.

Figure 3: The procedure of the Basic Global Thresholding.

2.3 Parallel application

Adding a smoothing mechanism with variable window sizes, six additional directions to the localization step and a global thresholding mechanism adds computational complexity to the original bidirectional Prewitt algorithm. These additional computations affect the overall performance. A parallel version of the proposed algorithm for the new shared memory MIMD multicore platforms is designed and implemented in order to speed up the computation. The parallel algorithm is designed and implemented using C/C++ as a base language and two open source libraries OpenMP and the open computer vision library (OpenCV) to overcome complexities added to the original algorithm. OpenCV library supports hundreds of optimized image processing algorithms mainly designed for real time applications contributing to the field of computer vision specifically [13, 15]. These libraries lend themselves well to parallel processing [13, 16]. One of the main challenges working on shared memory multiprocessors is the work distribution [1, 17]. Work, for good distribution, can be divided into rows, columns or blocks. Using appropriate mechanisms for data partitioning not only provides an independent chunk of data that can be processed concurrently but also will add flexibility in tuning the algorithm to run more efficiently on different architectural platforms to enhance data locality that in turn reduces the number of time-consuming cache misses. Work distribution starts by allowing each of the processors to copy its assigned private data to its local memory. To gain efficiency, it is desired to decrease the number of times needed to copy data from slower shared memory to local memory at each computational unit. The algorithm starts dividing the image into a number of equal sized tiles (sub-images). Parallel processes will work independently on different sub-images using a self-scheduling technique for work distribution. Each processor applies smoothing, localization, and iterative thresholding before writing the processed data to its final location as shown in Figure 4.

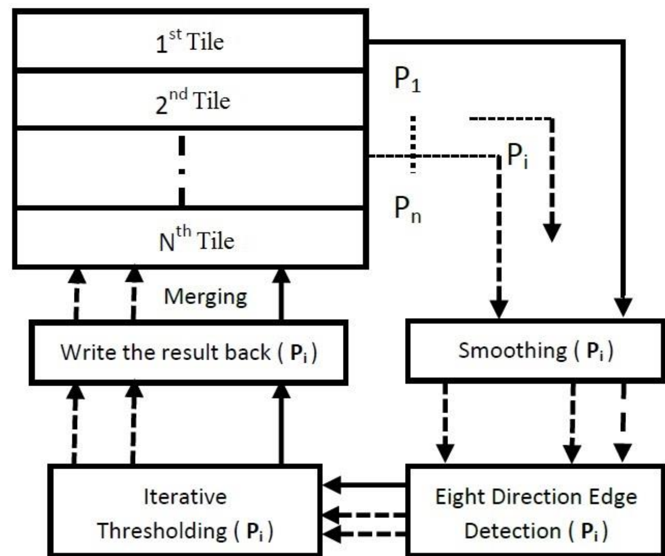

Figure 4: Processing life cycle.

As seen in Figure 4, the parallelism is applied at the highest possible level in which all work, in between the time of loading the main image up until merging sub-images, is done in parallel. We are presenting only partitioning into rows only. Yet our implementation supports all three partitioning methods (rows, columns, and blocks). The original input image is defined as an object (Region Of Interest, ROI) in the OpenCV library resulting in the need for some synchronization when the final result is written back to this object. This is a requirement enforced by OpenCV to guarantee data integrity, therefore, each processor will write the data to its original region of interest atomically. It is best to choose self-scheduling instead of pre-scheduling for good work balance. This will prevent the problem of having some processors idle while the others have excess work due to varying amount work required in each region of the image. It is important to note that sub-images are completely treated individually and are processed within the iterative thresholding mechanism separately. This strategy, according to our experiments, can result in better detection where more real edges are detected. Self-scheduling will overcome the difference in time required by applying the iterative thresholding mechanism on the assigned data as the number of computations may vary depending on the data itself.

LBP-based Hierarchical Sparse Patch Learning for Face Recognition

Yue Zhao[1], **and Jianbo Su**[1]
[1]Department of Automation, Shanghai Jiao Tong University,
and Key Laboratory of System Control and Information Processing,
Ministry of Education, Shanghai, 200240, China

Abstract— *Local Binary Pattern (LBP) features and its variants are computed on the patches with the fixed positions and a fixed size in images, while the limited variety of the size and position cannot accurately measure the nature of face image. In this paper, we propose a new learning method, Hierarchical Sparse Patch Learning (HSPL), to select face patches with different positions and sizes for face recognition. HSPL employs a sparse learning model to hierarchically select patches at two levels: in level 1 the optimal patch candidates are figured out, while in level 2 the optimal patches from the candidates are obtained. LBP features are extracted from the optimal patches to recognize faces. Experimental results show that the proposed method is more efficient and achieves higher recognition rate than the other two compared methods.*

Keywords: Feature Description, Face Recognition, Patch Selection, LBP, Sparse Learning

1. Introduction

Face feature description plays an essential role for face recognition[1]. In recent years, many face feature description algorithms have been proposed, including Eigenface [2], Fisherface [3], Local Binary Pattern (LBP) [4] and Elastic Bunch Graph Matching [5]. Among them, LBP and its variants [4], [6], [7], [8], [9] are the most widely used face feature descriptors, which divide the face image into patches with the fixed positions and a fixed size to extract LBP features [10]. However, the way that LBP and its variants adopted for patch generation leads to limited variety of the size and position of the obtained features. In order to solve this problem, Zhang *et al.* [11] scan the face image with a scalable sub-window, and over 7000 sub-patches are obtained. Then they use Adaboost learning algorithm to select an optimal subset of local patches and extract LBP features from these patches.

By shifting and scaling a sub-window, abundant patches with different positions and sizes could be generated with Zhang's method [11]. The features extracted from these patches yield a more complete and agile description of the face image. Unfortunately, there are two problems of the application of Adaboost learning algorithm in Zhang's method. Firstly, the training process and the test process are not accordant. The training process of Adaboost is iterative, and each cycle selects only one patch. For any two patches generated in the neighboring cycles, the former is preferable to the latter for face recognition. But in the test process, all the patches are treated equally, which leads to inferior combination of selected patches for recognition. Secondly, the method is inefficient, since the optimal patch should be selected via comparisons among all patches in each cycle.

In order to attenuate these two problems, this paper proposes a new learning method, Hierarchical Sparse Patch Learning (HSPL), to select adaptive face patches, and then to recognize faces. Firstly, it proposes a feature transformation to generate within- and between-class distance vectors, and constructs a sparse learning model based on them, which can automatically select adaptive patches. Based on the sparse learning model, HSPL hierarchically select patches at two levels: in level 1 patch candidates are obtained by automatical parameter setting, while in level 2 the optimal patches from the candidates are reached.

The rest of this paper is organized as follows. The proposed method is presented in Section 2. In Section 3, some experiments are performed to evaluate the performance of the proposed method, followed by conclusions in Section 4.

2. Hierarchical Sparse Patch Learning

2.1 Sparse Learning Model for Adaptive Patch Selection

2.1.1 Within- and Between-class Distance Vectors Generation

Given any two face images I and $\tilde{I}$, if we shift and scale a sub-window on them, respectively, we can get N adaptive patches for each image with different positions and sizes [11]. $I = [T_1, \dots, T_j, \dots, T_N]$ and $\tilde{I} = [\tilde{T}_1, \dots, \tilde{T}_j, \dots, \tilde{T}_N]$, where T_j and $\tilde{T}_j$ are any one patch from I and $\tilde{I}$. After extracted LBP features [4] from T_j and $\tilde{T}_j$, two histogram feature vectors are obtained as $T_j = [t_1, \dots, t_k, \dots, t_l]$ and $\tilde{T}_j = [\tilde{t}_1, \dots, \tilde{t}_k, \dots, \tilde{t}_l]$, where T_j and $\tilde{T}_j$ are both a $1 \times l$ histogram feature vector. t_k and $\tilde{t}_k$ are the kth element of T_j and $\tilde{T}_j$ respectively.

Procedure of the proposed algorithm:

1) **Initialization**
  a) Input Image
  b) User specifies the following options:
    i) Partitioning Method: rows, columns, or blocks (due to the space constrain, we will demonstrate row partitioning only)
    ii) Number of desired sub images, [numOfSubImage];
    iii) Number of processes, [numOfProcl];
    iv) Enabled or disabled: Smoothing technique, Thresholding. [toSmooth],[toThreshold];
    v) Noise level of the image (visual guess): 1 very low – 5 very high. [levelOfNoise];
2) **Partitioning data**
  a) Divide into the sub work as specified by the user's chosen mechanism of partitioning.
  b) Divide the work among the assigned processors. Processes will self-schedule to obtain the next available work
3) **On each of the sub works (if any) do the following:**
  a) De-noising
    i) If smoothing is enabled :
      (1) Apply improved median filter with specific window size on the image.
      (2) The size of the window depends on the visual guess specified by the user at 1.b.v. (Noise Level)
  b) Localization
      Eight-direction Prewitt (as described in Section 1)
  c) Thresholding
    i) If thresholding is enabled then apply the basic iterative global thresholding (as described in Section 2.2, Figure 3)
  d) Merging data
    i) Merge the processed sub work to its final destination.
  e) Return processed image

Pseudocode of partitioning into rows (our implementation supports rows, columns and blocks partitioning mechanisms). The Pseudocode conventions are taken from [17]:

```
procedure parmain(initialization as specified in step 1)
    Img [imgHeight,imgWidth] := imRead[imgName];   /* Loading the image */
    workLoad := imgHeight/numOfSubImage;
    shared img, destImage, numOfSubImage, levelOfNoise, numOfProc, toThreshold,toSmooth, workLoad, imgWidth, imgHeight;
    private i, subImg,x,y;
    Self-Scheduled forall i := 0 until numOfSubImage  /* Dynamic Scheduling of parallel processes */
    begin    /* partitioning data into sub-images each having the same imgWidth but with specific height*/
        x : = 0;
        y := i * workLoad;          /* We are giving an example of partitioning into rows */
        subImage := Rectangle( Img, Rect(x, y, imgWidth, imgHeight/numOfSubImage)); /* copy specified rect to subImage */
        if (toSmooth) then                /* De-noising: if smoothing is enabled */
            subImage := Smoothing (subImage,LevelOfNoise);
        end      /* Localization: Eighth direction Prewitt */
        subImage := eightDirections (subImage);
        if (Thresholding) then     /* Thresholding:  if thresholding is enabled */
            subImage := iterativeThresholding (subImage);
        end
        critical work;       /* Merging Data: /* Writing Data to its final destination*/
            mergeSubImage (destImage, subImage, Rect (x, y, imgWidth, imgHeight /numOfSubImage ));
        end critical;
            Release(subImage);
    end
    Return (destImage);       /* Return Processed image*/
End Procedure
```

Figure 5: Pseudocode of the proposed algorithm.

3 Experimental results and analysis

To compare the quality of the proposed Improved Median filter that uses variable-sized windows with the standard Median that uses a fixed window of size 3x3, we use the well-known Peak Signal-to-Noise Ratio PSNR (dB) [2]. The results are shown in Table 1 in which the test image (Lena see, Figure 6) is corrupted with varying degrees of impulse noise. As shown in Table 1, the improved Median filter generates significantly higher quality output than the standard in every case. The comparison of various window sizes and various noise levels and types can be seen from the first (input image indicating noise type and level) and forth (Median filter and the corresponding window size) columns of Figure 6. Each row of the figure represents application of different algorithms to the input image in column 1. To visually demonstrate the quality of the resulting image, using PSNR measurement, the output of the Canny (industry standard) is shown in Figure 6.

"Standard Canny" refers to the default run with Matlab 7.10.0 (R2010a) without any tuning (Column 2) while "best Canny" is the best results (Column 3) that can be produced by tuning the supported parameters.

From Figure 6 in the following page, we can see the capability of the proposed algorithm to suppress Salt & Pepper noise even when the image is impacted with very high noise percentage. Furthermore, the algorithm is able to suppress different types of noise with less impact on the real edges than the bidirectional Prewitt implemented with Matlab.

Table 1: Comparison of quality between standard Median filter and our proposed one using PSNR applied on Lena impacted with different percentage of impulse noise

Image: Lena 512 x 512, (see Figure 6 for image)				
Impulse Noise Percentage	Standard Median filter PSNR(dB)	Improved Median filter		
		PSNR (dB)	Window size	Noise level
10	33.5798	33.5798	3x3	level 1
20	29.5107	30.2259	5x5	level 2
30	24.1262	29.3523	5x5	level 2
40	19.1859	27.8228	5x5	level 2
50	15.3806	26.646	7x7	level 3
60	12.4786	25.2654	9x9	level 4
70	10.098	23.2942	Double 9x9	level 5

In order to provide a fair comparison of the efficiency of our algorithms and implementation choices, we compare the sequential execution times of our proposed method with the existing implementations of bidirectional Prewitt and Canny algorithms with Matlab that are already in-use in Table 2. Table 2 shows the results of all tests-cases run sequentially on the same platform. As mentioned in the previous section, our algorithms are implemented using C/C++ as the base language with OpenCV.

Table 2: Sequential run of the presented algorithm vs. the bidirectional Prewitt and Canny edge detections implemented within Matlab

Image Details		Intel I Core ™ i7-3720QM CPU @ 2.60 GHz, 16 GB RAM, 64-bit Windows 7, L1 D :4 x 32 Kbytes 8-way set associative, L1 I :4 x 32 Kbytes, 8-way set associative L2 : 4 x 256 Kbytes 8-way set, associative, L3 : 6 Mbytes 12-way set associative			
Image	Size	Proposed 8 Direction Prewitt by itself	Proposed: Smoothing (level1) & 8 Direction Prewitt & Iterative Thresholding	Matlab Bidirectional Prewitt	Matlab Canny Edge detection
15315 x 11624	169 MB	2.456 sec	3.51 sec	8.719067 sec	55.23517 sec
2250 x 2250	14.4 MB	0.063 sec	0.171 sec	0.8991 sec	2.411595 sec
1536 x 2048	5.91 MB	0.047 sec	0.109 sec	0.263233 sec	1.494402 sec
512 x 512	768 KB	0.0145 sec	0.016 sec	0.137101 sec	0.334044 sec
256 x 256	192 KB	0.001 sec	0.0025 sec	0.026171 sec	0.2432 sec

As shown in Table 2, our sequential implementation outperforms bidirectional Prewitt and Canny edge detections. It is important to note that in the table, execution times reported for our implementation include smoothing (with different levels), localization and the iterative thresholding technique. From observing Table 2 (running on Intel i7) processing an image of dimension 512 x 512 takes 16 ms resulting in 62.5 fps (Frames per Second) with level 1 smoothing [43 ms (23.2 fps) when it is smoothed with level 5]. Obviously higher noise percentage requires more computations for good suppression. However, we are able to accelerate the previous runs using 4 processors, running on the same platform, to 6 ms at 166 fps for level 1 smoothing [and 11 ms at 90.9 fps for level 5]. Having shown how efficient the algorithm works on images of small dimensions, we present the execution times of our parallel implementation working on larger images using two different multiple core platforms as its shows in Table 3 and Table 4 respectively. Due to space constraints a subset of the general cases are presented. The two multicore platforms used for our experiments are a 12-core AMD with Opteron module (Table 3) and a 64-core AMD with Bulldozer module (Table 4). The organizations of the two processors are quite different. The Opteron consists of two chips each with 6 cores each. Each Opteron core has its own private first level instruction and data caches of 64 KB size each and a private unified L2 cache of 512 KB. Every 6 core/chip share 6 MB of L3 cache. On the other hand, the 64-core AMD with Bulldozer consists of eight Bulldozer modules on one chip (16 cores). A Bulldozer module consists of two integer execution cores that share a first level instruction cache of 64 KB and a floating point unit. Each core has a private first level data cache of 16 KB. The two cores within one Bulldozer module share a unified L2 cache (2 MB). Every four modules share 16 MB of L3 cache.

The L1 data cache in Opteron is four times larger than the Bulldozer's L1 data cache. The idea of partitioning the image into a number of equal sized tiles helps enhance the locality of data. The main difference between Opteron and Bulldozer is that more resources are shared within the Bulldozer module. For instance, when we process relatively large images (29649 x 22008) using 32 cores, running on the Bulldozer, the best results are achieved when the parallel task is distributed such that each task is executed on every other core. This means the 32 Bulldozer modules, one active core per module, will be working compared to the case in which only 16 Bulldozer modules, two active cores per unit, are used. Using more Bulldozer units not only provides larger L3 cache, a total of total 128 MB on the four chips, but also allows each core to use the entire shared L2 cache as a dedicated second level cache and have exclusive use of the single floating point unit that is shared by the two integer cores. Although the communication is costly especially if the cores do not have shared cache space, i.e., the cost of one core accessing cache memory located on different chips, having a better data locality (when larger caches are used) enhances the overall performance of the application. This statement will not apply to all applications, but in our design we aimed to divide the input image to independent tiles in order to decrease the amount of communication required at the same time increases the locality of data gained from using larger cache space.

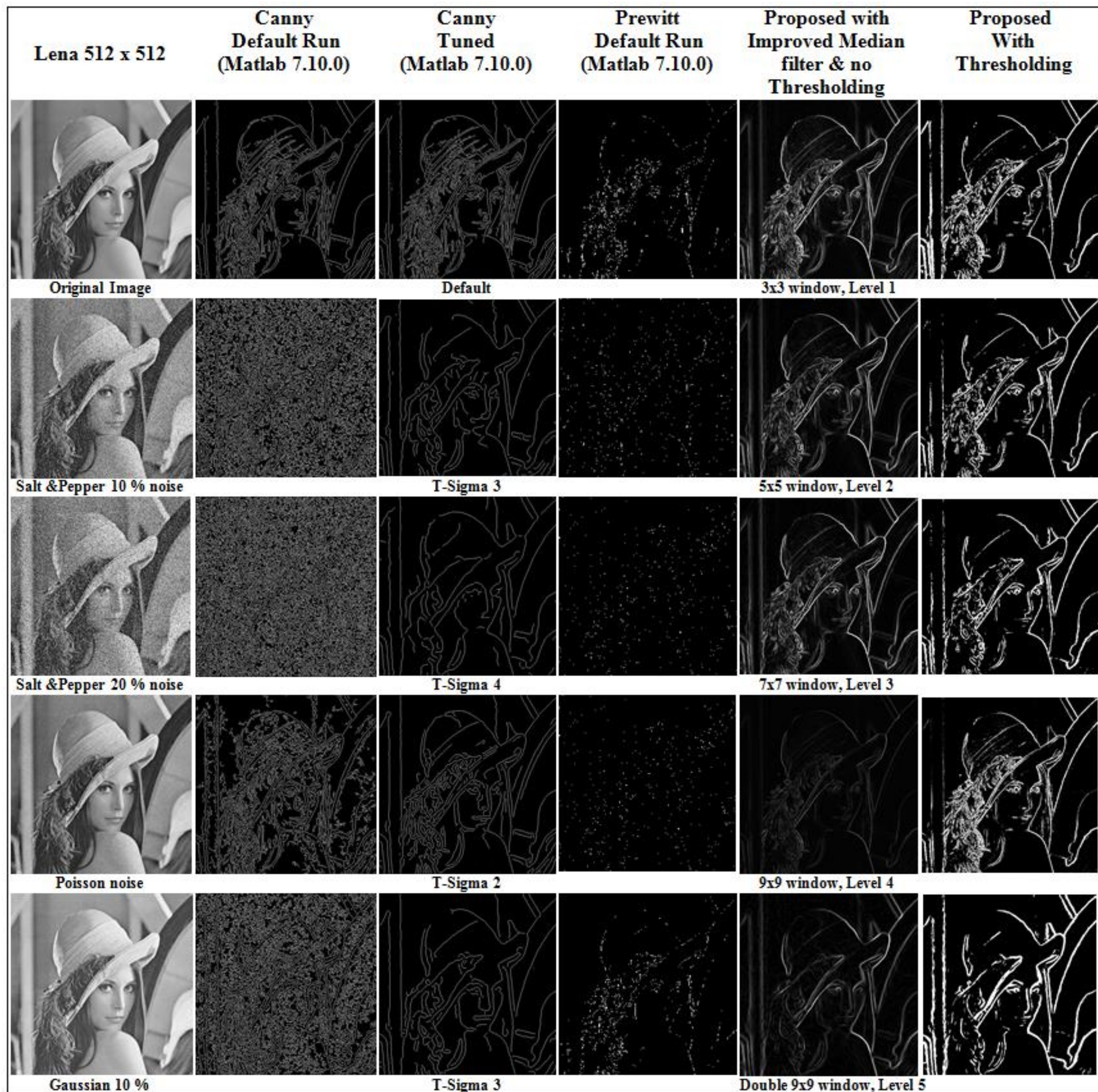

Figure 6: Output of Canny (industry standard) and bidirectional Prewitt vs. our proposed work

Table 3: Execution time for the proposed algorithm running on 12-core compute nodes

Case 1: Eight direction Prewitt, Case 2: Smoothing (level 1) & Eight direction Prewitt, Case 3: Smoothing (level 1) & Eight direction Prewitt & thresholding, Case 4: Smoothing (level 5) & Eight direction Prewitt & thresholding								
		Image of size 29649 x 22008				**Image of size 15315 x 11624**		
	N cores	**Best execution time T_N (sec)**	**Sequential T_1 (sec)**	**Speed up**	**Number of cores**	**Best execution time T_N (sec)**	**Sequential T_1 (sec)**	**Speed up**
Case 1	12	2.890	22.850	7.91	12	0.803	6.021	7.50
Case 2	12	3.145	23.798	7.57	12	0.843	6.534	7.75
Case 3	12	4.521	33.532	7.42	12	1.381	8.854	6.41
Case 4	12	13.015	143.52	11.03	12	3.724	39.45	10.59

		Image of size 2250 x 2250				Image of size 1536 x 2048		
Case 1	12	0.028	0.180	6.43	8	0.024	0.108	4.50
Case 2	12	0.032	0.189	5.91	8	0.026	0.114	4.38
Case 3	12	0.067	0.491	7.33	12	0.051	0.305	5.98
Case 4	12	0.147	1.589	10.81	12	0.130	1.016	7.82
		Image of size 2048 x 2048				**Image of size 2704 x 4064**		
Case 1	8	0.031	0.147	4.74	8	0.069	0.374	5.42
Case 2	8	0.031	0.156	5.03	8	0.075	0.403	5.39
Case 3	12	0.051	0.372	7.29	12	0.160	1.297	8.11
Case 4	12	0.132	1.201	9.10	12	0.348	3.959	11.38
12-core compute node (shared memory multi-processor, processor @ 2.2 Ghz of AMD (Opteron) type, 6x64 KB L1 instruction cache per processor, 6x64 KB L1 data cache per processor, 6x512 KB L2 per processor, 6MB L3 per processor, 24 GB RAM, 64-bit Linux version 2.6.18.								

Table 4: Execution time for the proposed algorithm running on 64-core compute nodes

Case 1: Eight direction Prewitt, Case 2: Smoothing (level 1) & Eight direction Prewitt, Case 3: Smoothing (level 1) & Eight direction Prewitt & thresholding, Case 4: Smoothing (level 5) & Eight direction Prewitt & thresholding								
		Image of size 29649 x 22008				**Image of size 15315 x 11624**		
	N cores	**Best execution time T_N (sec)**	**Sequential T_1 (sec)**	**Speed up**	**Number of cores**	**Best execution time T_N (sec)**	**Sequential T_1 (sec)**	**Speed up**
Case 1	28	1.35	19.07	14.13	28	0.39	5.22	13.38
Case 2	32	1.35	19.89	14.73	24	0.42	5.46	13.00
Case 3	32	1.86	28.14	15.13	32	0.65	7.64	11.75
Case 4	44	4.35	115.56	26.57	40	1.32	31.41	23.80
		Image of size 2250 x 2250				**Image of size 1536 x 2048**		
Case 1	12	0.027	0.155	5.74	12	0.019	0.096	5.05
Case 2	12	0.029	0.163	5.62	12	0.023	0.102	4.43
Case 3	16	0.051	0.460	9.02	16	0.040	0.284	7.10
Case 4	24	0.094	1.461	15.54	24	0.073	0.979	13.41
		Image of size 2048 x 2048				**Image of size 2704 x 4064**		
Case 1	12	0.023	0.126	5.48	16	0.040	0.324	8.10
Case 2	12	0.025	0.139	5.56	16	0.043	0.342	7.95
Case 3	12	0.043	0.336	7.81	24	0.117	1.135	9.70
Case 4	24	0.092	1.067	11.60	32	0.184	3.364	18.28
64-core compute node (shared memory multi-processor, processor @ 2.2 Ghz of AMD (Bulldozer) type, 1x64 KB L1 instruction cache per module which contains two execution cores, 2x16 KB L1 data cache per module, 1x2 MB L2 per module, 16 MB L3 shared by four modules (located on the same chip), 128 GB RAM, 64-bit Linux version 2.6.18.								

4 Conclusion

A parallel edge detection application based on eight direction Prewitt edge detection algorithm is designed and implemented to work on different multicore platforms efficiently. Different functionalities are added to the original Prewitt such as smoothing and a global thresholding mechanism. In order to suppress noise more efficiently, an improved Median filter that enables the application to work effectively on noisy images is added. This method not only strengthens the original algorithm by allowing it to work on noisy images more effectively but also lets it compete with the industry standard detection algorithm Canny. Our algorithm when run sequentially, with all added functionality and complexity included, outperforms the default runs of both Prewitt and Canny already implemented in Matlab. Our parallel implementation of the algorithm uses C/C++ as the base language with two open source libraries OpenCV and OpenMP. Different experiments show improved performance gained from processing different size images especially when the complexity of the problem increases. Variety of tuning mechanisms have been added throughout the design to allow flexibility of work distribution to enhance the overall performance. The parallel implementation of this application is tested on two new shared memory MIMD multicore platforms namely Opteron and Bulldozer. Finally this implementation can effectively be used within the applications of image processing that relies on fast and accurate edge detection.