

# PARALLEL GENERATION OF IMAGE LAYERS CONSTRUCTED BY EDGE DETECTION USING MESSAGE PASSING INTERFACE

Alaa Ismail Elnashar

Faculty of Science, Computer Science Department, Minia University, Egypt  
Associate professor, Department of Computer Science, College of Computers and  
Information Technology, Taif University, Saudi Arabia

## **ABSTRACT**

*Edge detection is one of the most fundamental algorithms in digital image processing. Many algorithms have been implemented to construct image layers extracted from the original image based on selecting threshold parameters. Changing these parameters to get a high quality layer is time consuming. In this paper, we propose two parallel techniques, NASHT1 and NASHT2, to generate multiple layers of an input image automatically to enable the image tester to select the highest quality detected edges. In addition, the effect of intensive I/O operations and the number of parallel running processes on the performance of the proposed techniques have also been studied.*

## **KEYWORDS**

*Parallel programming, Message Passing Interface, performance, Image Processing, Edge Detection*

## **1. INTRODUCTION**

A digital image can be represented by a two-dimensional array having integer values. Color digital images consist of a set of pixels; each pixel represents a combination of primary colors. A channel is the grayscale image of the same size as a color image, made of just one of these primary colors. A digital color image has three channels; red, green and blue while a grayscale image has just one channel.

Digital image processing is the use of computer algorithms to perform image processing on digital images. It has many advantages over analog image processing. It allows a much wider range of algorithms to be applied to the input data and can avoid problems such as the build-up of noise and signal distortion during processing. It also offers high performance using simple tasks, and can implement methods which would be impossible by analog means.

Edge detection is a set of mathematical methods which aim at identifying points in a digital image at which the image brightness changes sharply or has discontinuities. The points at which image brightness changes sharply are typically organized into a set of curved line segments termed edges. An edge within an image can be defined as discontinuities in image intensity from one pixel to another. Edge detection is a fundamental tool in image processing, machine vision and computer vision, particularly in the areas of feature detection and feature extraction [1].

Many algorithms [2, 3, 4, 5, 6] were designed and implemented to capture such edges. The purpose of edge detection algorithms is to reduce the amount of data in an image, while preserving the structural properties to be used for further image processing .

In the ideal case, the result of applying an edge detector to an image may lead to a set of connected curves that indicate the boundaries of objects, the boundaries of surface markings as well as curves that correspond to discontinuities in surface orientation. Thus, applying an edge detection algorithm to an image may significantly reduce the amount of data to be processed and may therefore filter out information that may be regarded as less relevant, while preserving the important structural properties of an image.

If the edge detection step is successful, the subsequent task of interpreting the information contents in the original image may therefore be substantially simplified. However, it is not always possible to obtain such ideal edges from real life images of moderate complexity [7].

Threshold is one of the most widely used methods for image segmentation. It is useful in discriminating foreground from the background by selecting an adequate threshold value [8]. Several threshold techniques [9, 10, 11, 12, 13] have been implemented to adapt threshold values to produce high quality edges. Some studies discussed edge detection parallelization to increase the execution speedup [14, 15, 16, 17].

In [18] a comparison between loop-level parallelism and domain decomposition is presented. Christos et al implemented a real-time edge detection parallel technique for FPGAs [19].

A real-time interactive image processing parallel system designed for manipulating large size images is described in [20]. An inherent parallel scheme for 3D image segmentation of large volume data on a GPU cluster is presented in [21].

Canny edge detector [26] has been implemented using CUDA system and achieved 50 times speedup compared with CPU system [22]. Prakash et al proposed a technique that uses multi-cores and GPU implementation through MPI and OpenCL to perform edge detection process for multiple images in parallel [23].

A parallel framework for image segmentation using region based techniques is presented in [24]. The algorithm is based on performing several segmentations of the same image using a parallel region-based algorithm. Moreover, these segmentations are also obtained in parallel.

All the mentioned studies focus on either image partitioning or parallelizing edge detection components such as smoothing and suppression.

In this paper we introduce two parallel techniques, NASHT1 and NASHT2 that apply edge detection to generate a set of layers for an input image. In addition, we study the effect of the number of parallel running processes and image size on the performance of the proposed techniques.

The paper is organized as follows: section 2 defines the research problem. In section 3, the ordinary sequential image detection technique is presented. Section 4 presents the proposed parallel techniques. The experiments carried out and results discussion are presented in Section 5.

## 2. PROBLEM DEFINITION

In edge detection process, the quality of the output image layer is very sensitive to the input parameters thresholds [11]. The main drawback of Canny edge detection is that  $T_{high}$ ,  $T_{low}$  and  $\sigma$  are set manually as input parameters, and it is possible to get a proper threshold after many experiments based on the quality of the output edges. However, in practice, the  $T_{high}$  and  $T_{low}$  often

change because the scenes and illumination change frequently [25]. So, the user choice is to change the input parameters several times to generate an image layer that satisfies his needs.

Another choice is to plug the image detector engine into an iterative loop to get a set of layers to select the required one(s) from this set. This solution may save time and effort consumed in manual process.

The disadvantage of this solution is the intensive I/O operations required to generate a large number of layers. This will lead to a dramatically execution time increase. The idea of our proposed parallel technique is to divide the iterative process of generating successive layers among several parallel processes to reduce the execution time.

### 3. SEQUENTIAL IMAGE LAYERS GENERATION TECHNIQUE

Canny algorithm [26] described in Figure 1, is one of the most famous edge detection algorithms. It uses two functions "Read" and "Write" to read the input image and to write out the generated layer, as shown in Figure 2 and Figure 3.

```

Algorithm Edge_Detector (Image_File_Name, Layer, Im_Rows, Im_Cols ,  $T_{high}, T_{low}$ 
,  $\sigma$ )

/* Input: Image_File_Name,  $T_{high}, T_{low}, \sigma$  */
/* Output: Imager Layer */
01. Read (Image_File_Name, Im_Rows , Im_Cols);
02. Apply Gaussian smoothing on the image using the input standard deviation  $\sigma$  ;
03. Compute the first derivative in the x and y directions;
04. Compute the magnitude of the gradient;
05. Perform non-maximal suppression;
06. Mark the edge pixels using  $T_{high}$  and  $T_{low}$  ;
07. Form Layer_File_Name based on  $T_{high}, T_{low}$  and  $\sigma$  ;
08. Call Write (Layer_File_Name, Layer, Im_Rows , Im_Cols);
09. End
    
```

Figure 1: Canny Edge Detector Pseudo Code

```

Read(Image_File_Name, Image, Im_Rows , Im_Cols)
/* Input: Image_File_Name */
/* Output: image array Image, Im_Rows , Im_Cols */
01. Allocate memory array Image to store the image data;.
02. Open (Image_File_Name) for read;
03. Read Image header;
04. Im_Rows= rows;
05. Im_Cols = columns;
06. for i = 1 to Im_Rows
07. {
08.   for j = 1 to Im_Cols
09.   {
10.     Read Image data;
11.     Image[i][j] = Image data;
12.   }
13. }
14. Close (Image_File_Name);
15. Return Image, Im_Rows , Im_Cols ;
16. End
    
```

Figure 2: Reading Image Pseudo Code

It computes the edges pixels based on specific statistics of the concerned image. In this algorithm, edge detection process can be affected by image noise, it is very important to filter out the noise to prevent false edge detection caused by noise. To smooth the image, a Gaussian filter [27],  $G$ , is applied to convolve with the image. This step smoothes the image to reduce the effects of noise by generating an array of smoothed data  $S[i, j] = G[i, j; \sigma] * I[i, j]$ , where  $I[i, j]$  denotes an image of size  $i \times j$  and  $\sigma$  is the value of standard deviation that is used in image smoothing process [28]. The algorithm computes the first derivative in the x and y directions and finds the magnitude of the gradient.

Non-maximum suppression for the gradient magnitude is then applied. Two input parameters  $T_{high}$  and  $T_{low}$  are used to detect and connect edges. Pixels with values greater than  $T_{high}$  are assigned the binary value 1 in the output, while pixels with values below  $T_{low}$  are assigned the binary value 0. Pixels with values between  $T_{low}$  and  $T_{high}$  are assigned the binary value 1 in the binary output if they can be connected to any pixel with a value larger than  $T_{high}$  through a chain of other pixels with values larger than  $T_{low}$ . Finally, the algorithm writes out the edge image to the output image layer file.

Algorithm 1 generates only one image layer based on the input values for  $T_{high}, T_{low}$  and  $\sigma$ . The algorithm can be invoked inside an iterative loop that modifies these values to generate multiple image layers as shown in Figure 4.

```

Write (Layer_File_Name, Layer, Layer_File, Im_Rows , Im_Cols)
/* Input: Layer_File_Name, Layer array, Im_Rows , Im_Cols */
/* Output: Image Layer File */
01. Open (Layer_File_Name) for write;
02. write (Layer_File_Name) Im_Rows , Im_Cols;
03. for i = 1 to Im_Rows
04. {
05.  for j = 1 to Im_Cols
06.  {
07.   write (Layer_File_Name) Layer[i][j];
08.  }
09. }
10. Close (Layer_File_Name);
11. End

```

Figure 3: Writing Image Layer Pseudo Code

```

Sequential Iterative Layers Generator (Image_File_Name, Layer, Im_Rows, Im_Cols
                                     ,  $T_{high}, T_{low}$  ,  $\sigma$  , Number_Of_Layers)

/*Input: Image_File_Name, initial values of  $T_{high}, T_{low}$  and  $\sigma$  , Number_Of_Layers */
/* Output: Multiple Image Layers */
01. for i = 1 to Number_Of_Layers;
02. {
03.  Edge_Detector (Image_File_Name, Layer, Im_Rows, Im_Cols ,  $T_{high}, T_{low}$  ,  $\sigma$  )
04.  Update  $T_{high}, T_{low}$  ,  $\sigma$  according to loop counter;
05. }
06. Close (Layer_File_Name);
07. End

```

Figure 4: Sequential Iterative Layers Generator Pseudo Code

## 4. PROPOSED PARALLEL TECHNIQUES

Two parallel techniques with four versions were designed and implemented. The first goal of this paper is to parallelize layers generation process by distributing layers generation iterations chunks among several parallel processes. The second goal is to study the execution time reduction gained from parallelization. The last goal is to study how the number of running parallel processes and image size affect proposed techniques relative speedup.

### 4.1 PARALLEL PLATFORM AND SOFTWARE

We used an experimental system consisting of the following hardware and software components:

- 1- PC1: Intel® Core i7 CPU @ 2.40GHz, 6.00 GB RAM, running on Windows 10 Pro.
- 2- PC2: Intel® Core i5 CPU @ 2.40GHz, 4.00 GB RAM, running on Windows 7 Ultimate.
- 3- PC3: Intel® Core i5 CPU @ 2.30GHz, 4.00 GB RAM, running on Windows 10 Pro.
- 4- Gigaset® SE551 WLAN Ethernet adaptor with LAN cables.
- 5- MicroSoft® Visual C++ 2010 Compiler.
- 6- MPICH2 for Microsoft Windows.
- 7- Jumpshot-4 visualization software.

Three personal computers PC1, PC2 and PC3 with the above specifications are selected randomly to carry out running and evaluation experiments. For individuals, finding a set of such heterogeneous systems is easier than finding a set of homogenous devices to be used in real work environment. The three PCs are connected to each others by using an Ethernet adaptor as described in figure 5. The proposed techniques were coded and compiled by using MicroSoft® Visual C++ 2010 Compiler.

MPICH2 [29] is a high performance portable implementation of Message Passing Interface. It efficiently supports different computation and communication platforms. It also supports using of C/C++ and FORTRAN programming languages.

The structure of MPICH2 is shown in figure 6. It uses an external process manager for scalable startup of MPI jobs. The default process manager is called MPD, which is a ring of daemons on the machines where MPI programs run. The CH3 device contains different internal communication options called "channels". "Socket channel" is the traditional TCP sockets based communication channel. It uses TCP/IP sockets for all communication including intra-node communication [30].

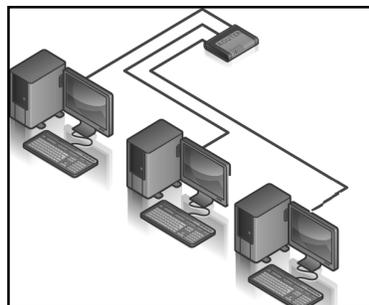


Figure 5. Hardware Platform

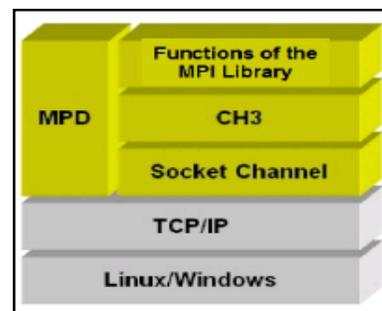


Figure 6. MPICH2 Structure

In contrast to MPICH2 for Windows, the implementation for UNIX and LINUX offers built-in network topology support. This makes an easy use of MPICH2 on such platforms and hence little attention has been focused on using the implementation on Microsoft Windows although it provides the facilities of parallel execution and multi-threading.

MPICH2 for windows can be installed either on a single machine having single / multi-core processors or an interconnected set of machines. In both cases, performance of MPI programs is affected with various parameters such the number of cores (machines), number of running processes and the programming paradigm which is used [31]. MPI offers two patterns of inter-process communication, Point-to-point and Collective communication.

In point-to-point communication, a message is transmitted from a process to another one using a buffer. After the data is packed into a buffer, the communication device is responsible for routing the message to the proper location. The location of the message is defined by the process's rank.

Collective communication pattern is defined as communication that involves a group of processes. MPI implements this pattern by using MPI\_Bcast function which broadcasts a message from the root process to all other processes including itself. This implies that the amount of data sent must be equal to the amount received, pair wise between each process and the root. It blocks until all processes have made a matching call to MPI\_Bcast, at which point communication occurs and execution continues.

To profile running executables and observe parallel processes communication graphically, we use Jumpshot-4 [32]; a visualization software for the logfile format, SLOG-2, [33]. The graphical symbols used in this paper with their icons, names and descriptors are shown in Table 1.

Table 1. Jumpshot-4 graphical symbols

Symbol	Name	Description
	Message	Message sent from one process to another one is represented by an arrow directed from source process to destination one.
	MPI_Barrier	Yellow state, blocks until all processes have reached this routine.
	MPI_Bcast	Aqua state, broadcasts a message from the process with rank "root" to all other processes
	MPI_Send	Blue state, sends a data message from a source process to a destination one.
	MPI_Receive	Green state, receives a data message from a source process.
	MPI_Finalize	Terminates MPI execution environment.

## 4.2 PARALLEL IMPLEMENTATIONS

Based on point-to point and collective communication patterns, we have implemented the proposed techniques, NASHT1 and NASHT2. Two versions were developed for each technique. Version1 implementation uses point-to-point communication pattern employing MPI "send" and "receive" functions. Version2 uses collective communication pattern employing "MPI\_Bcast" functions.

### 4.2.1 PARALLEL TECHNIQUE1, NASHT1: SEND VERSION

In this technique, the root process with rank "identifier" Zero, is designated to read the contents of an input image and then sends its data to all other processes. All processes except the root process compute their iteration chunks and then perform edge detection task based on their own local

values of  $T_{high}$ ,  $T_{low}$  and  $\sigma$ . Once this task is finished, the generated image layers are sent to the root process to be written. The steps of this technique are described in figure 7. A jump-shot time line for inter-process communication is shown in Figure 8.

```

01. Initialize MPI environment;
02. Initialize edge detection parameters  $T_{high}$ ,  $T_{low}$  and  $\sigma$ ;
03. Read the number of layers (N_layers) to be generated;
04. Determine the number of MPI processes (Npr) and their ids;
/*Determine the number of generated layers (N_layers_Pr) for each process */
05. N_layers_Pr = N_layers / (Npr - 1) ;
06. If id=master then
07.   Read(Image_File_Name, Image, Im_Rows , Im_Cols);
08.   Send Image rows (Im_Rows) to all processes;
09.   Send Image columns (Im_Cols) to all processes;
10.   Send Image array to all processes;
11.   Receive the output Layer_File_Name from any other process ;
12.   Receive Layer contents from any other process ;
13.   Write (Layer_File_Name, Layer, Layer_File, Im_Rows , Im_Cols);
14. Else
/* All processes except "master" execute the following part */
15.   Receive Image rows (Im_Rows) from master process;
16.   Receive Image columns (Im_Cols) from master process;
17.   Receive Image array from master process;
18.   Update edge detection parameters based on id value;
19.   for k = 1 to N_layers_Pr
20.   {
21.     Edge_Detector (Image_File_Name, Layer, Im_Rows, Im_Cols,  $T_{high}$ ,  $T_{low}$  ,  $\sigma$  ) ;
22.     Form the output Layer_File_Name based on  $T_{high}$ ,  $T_{low}$  and  $\sigma$  ;
23.     Send the output Layer_File_Name to master process ;
24.     Send Layer array to master process ;
25.   }
26. EndIf /* If 1 */
26. Finalize MPI environment
27. End
    
```

Figure 7. Parallel Technique1, NASHT1: Send Version Pseudo Code

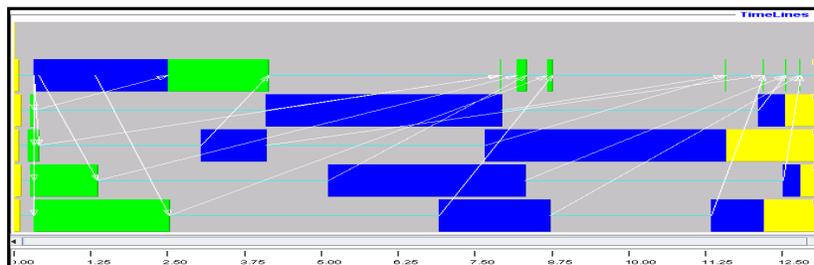


Figure 8. Parallel Technique1, NASHT1: Send Version Inter-Process Communication

#### 4.2.2 PARALLEL TECHNIQUE1, NASHT1: MPI\_BCAST VERSION

In this technique, the root process reads the contents of an input image. Then it broadcasts the image data to all other processes. Each process then computes its own iteration chunk and performs edge detection task based on its own local values of  $T_{high}$ ,  $T_{low}$  and  $\sigma$ . Once this task is

finished, the generated image layers are sent to the root process to be written. The steps of this technique are described in Figure 9. A jump-shot time line for inter-process communication is shown in Figure 10.

```

01. Initialize MPI environment;
02. Initialize edge detection parameters  $T_{high}, T_{low}$  and  $\sigma$ ;
03. Read the number of layers (N_layers) to be generated;
04. Determine the number of MPI processes (Npr) and their ids;
/*Determine the number of generated layers (N_layers_Pr) for each process */
05. N_layers_Pr = N_layers / Npr ;
06. If id=master then
07.   Read(Image_File_Name, Image, Im_Rows , Im_Cols);
08.   Receive the output Layer_File_Name from any other process ;
09.   Receive Layer array from any other process ;
10.   Write (Layer_File_Name, Layer, Layer_File, Im_Rows , Im_Cols)
11. EndIf
/* All processes execute the following part */
12. Master broadcasts Image rows (Im_Rows) to all processes;
13. Master broadcasts Image columns (Im_Cols) to all processes;
14. Master broadcasts Image array to all processes;
15. Update edge detection parameters based on id value;
16. for k = 1 to N_layers_Pr
17. {
18.   Edge_Detector (Image_File_Name, Layer, Im_Rows , Im_Cols,  $T_{high}, T_{low}$  ,  $\sigma$ ) ;
19.   Form the output Layer_File_Name based on  $T_{high}, T_{low}$  and  $\sigma$  ;
20.   Send the output Layer_File_Name to master process ;
21.   Send Layer array to master process ;
22. }
23. Finalize MPI environment
24. End
    
```

Figure 9. Parallel Technique1, NASHT1: MPI\_Bcast Version Pseudo Code

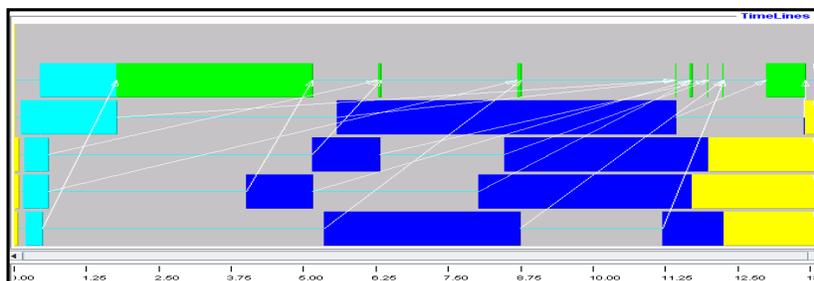


Figure 10. Parallel Technique1, NASHT1: MPI\_Bcast Version Inter-Process Communication

#### 4.2.3 PARALLEL TECHNIQUE2, NASHT2: SEND VERSION

In this technique, the root process reads the contents of an input image and then sends its data to the odd ranked processes which compute their iteration chunks and perform edge detection task based on their own local values of  $T_{high}, T_{low}$  and  $\sigma$ . Once this task is finished within each process, the generated image layers are sent to the neighbor even ranked process to be written. The root process writes its generated layers by itself. The steps of this technique are described in Figure 11. A jump-shot time line for inter-process communication is shown in Figure 12.

```

01. Initialize MPI environment;
02. Initialize edge detection parameters  $T_{high}, T_{low}$  and  $\sigma$ ;
03. Read the number of layers (N_layers) to be generated;
04. Determine the number of MPI processes (Npr) and their ids;
/*Determine the number of generated layers (N_layers_Pr) for each process */
05. N_layers_Pr = (N_layers) / ( int(0.5*Npr) + 1 ) ;
06. If id=master then /* If 1 */
07.  Read(Image_File_Name, Image, Im_Rows , Im_Cols);
08.  Send Image rows (Im_Rows) to all odd id processes;
09.  Send Image columns (Im_Cols) to all odd id processes;
10.  Send Image array to all odd id processes;
11. EndIf /* If 1 */
/* All odd id processes an master execute the following part */
12. If id is odd or master then /* If 2 */
13.  if id <> master then /* If 3 */
14.   Receive Image rows (Im_Rows) from master process;
15.   Receive Image columns (Im_Cols) from master process;
16.   Receive Image array from master process;
17.   Send Image rows (Im_Rows) to process with id equals id+1 ;
18.   Send Image columns (Im_Cols) to process with id equals id+1 ;
19. EndIf /* If 3 */
20. Update edge detection parameters based on id value;
21. for k = 1 to N_layers_Pr
22. {
23.  Edge_Detector (Image_File_Name, Layer, Im_Rows , Im_Cols,  $T_{high}, T_{low}$  ,  $\sigma$  ) ;
24.  Form the output Layer_File_Name based on  $T_{high}, T_{low}$  and  $\sigma$  ;
25. If id = master then /* If 4 */
26.  Write (Layer_File_Name, Layer, Layer_File, Im_Rows , Im_Cols); Else
27.  Send the output Layer file name and Layer array to process with id equals id+1 ;
28. EndIf /* If 4 */
29. }
30. EndIf /* If 2 */
31. If id is even then /* If 5 */
32.  Receive Image rows (Im_Rows) from process with id equals id-1 ;
33.  Receive Image columns (Im_Cols) from process with id equals id-1 ;
34.  Receive the output Layer file name from process with id equal id-1 ;
35.  Receive Layer contents from process with id equal id-1 ;
36.  Write (Layer_File_Name, Layer, Layer_File, Im_Rows , Im_Cols)
37. EndIf /* If 5 */
38. Finalize MPI environment
39. End

```

Figure 11. Parallel Technique2, NASHT2 : Send Version Pseudo Code

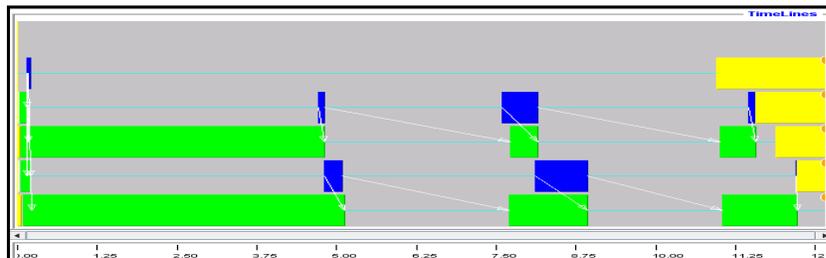


Figure 12. Parallel Technique2, NASHT2: Send Version Inter-Process Communication

**4.2.4 PARALLEL TECHNIQUE2, NASHT2: MPI\_BCAST VERSION**

In this technique, the root process reads the contents of an input image and broadcasts the image data to all other processes. Each process computes its iteration chunk and then performs edge detection task based on its own local values of  $T_{high}, T_{low}$  and  $\sigma$ . Once this task is finished within each process except the root process, the generated image layers are sent to the neighbor even ranked process to be written. The root process in this technique writes its generated layers directly after terminating edge detection task. The steps of this technique are described in Figure 13. A jump-shot time line for inter-process communication is shown in Figure 14.

```

01. Initialize MPI environment;
02. Initialize edge detection parameters  $T_{high}, T_{low}$  and  $\sigma$ ;
03. Read the number of layers (N_layers) to be generated;
04. Determine the number of MPI processes (Npr) and their ids;
/*Determine the number of generated layers (N_layers_Pr) for each process */
05. N_layers_Pr = (N_layers) / ( int(0.5*Npr) + 1 ) ;
06. If id=master then /* If 1 */
07.   Read(Image_File_Name, Image, Im_Rows , Im_Cols);
08. EndIf /* If 1 */
/* All processes execute the following part */
09. Master broadcasts Image rows (Im_Rows) to all processes;
10. Master broadcasts Image columns (Im_Cols) to all processes;
11. Master broadcasts Image array to all processes;
/* All odd id processes and master execute the following part */
12. If id is odd or master then /* If 2 */
13.   If id <> master then /* If 3 */
14.     Send Image rows (Im_Rows) to process with id equals id+1 ;
15.     Send Image columns (Im_Cols) to process with id equals id+1 ;
16.   EndIf /* If 3 */
17. Update edge detection parameters based on id value;
18. for k = 1 to N_layers_Pr
19. {
20.   Edge_Detector (Image_File_Name, Layer, Im_Rows , Im_Cols,  $T_{high}, T_{low}$  ,  $\sigma$  ) ;
21.   Form the output Layer_File_Name based on  $T_{high}, T_{low}$  and  $\sigma$  ;
22.   If id = master then /* If 4 */
23.     Write (Layer_File_Name, Layer, Layer_File, Im_Rows, Im_Cols);
24.   Else
25.     Send the output Layer_File_Name to process with id equals id+1 ;
26.     Send Layer array to process with id equals id+1 ;
27.   EndIf /* If 4 */
28. }
29. EndIf /* If 2 */
30. If id is even then /* If 5 */
31.   Receive the output Layer_File_Name from process with id equal id-1 ;
32.   Receive Layer array from process with id equal id-1 ;
33.   Write (Layer_File_Name, Layer, Layer_File, Im_Rows , Im_Cols);
34. EndIf /* If 5 */
35. Finalize MPI environment
36. End

```

Figure 13. Parallel Technique2, NASHT2: MPI\_Bcast Version Pseudo Code

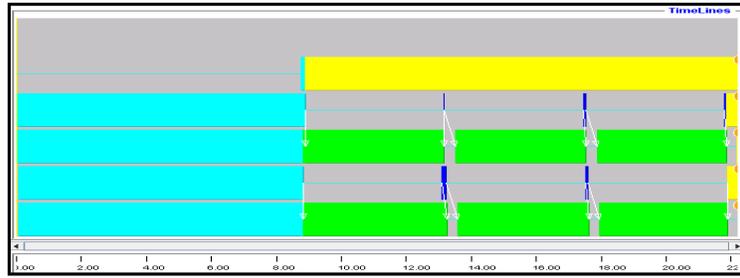


Figure 14. Parallel Technique2, NASHT2: MPI\_Bcast Version Inter-Process Communication

## 5. RESULTS AND DISCUSSION

We carried out some experiments to examine the achievement of the three goals of this paper. Regarding layers generation, we used three different images with various sizes ranging from small sized images to large sized ones. Small size, "Brain" image 165x 158, medium size, "Lena" image, 512 x 512 and large size "Picnic" image 1280 x 960.

The first experiment aims to examine the quality of the generated layers. All techniques were tested using the three images. The input images and samples of generated layers from test techniques are shown in Figure 15. The results show that there is no quality difference in the generated layers compared with the layer generated from sequential technique. The advantage of the proposed techniques demonstrated in this experiment is the generation of multiple layers in only one run instead of a single generated layer in case of sequential technique.

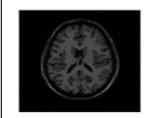
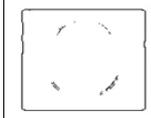
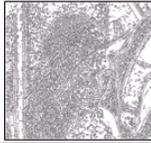
Original Image	Generated layers sample			
Brain 165 x 158				
				
Lena 512 x 512				
				
Picnic 1280 x 960				
				

Figure 15. Input images and samples of generated layers

Experiment 2 was carried out to study whether any execution time reduction is gained from parallelization. The experiment was designed to generate 720 layers for the input image in parallel.

Table 2 shows the serial execution time and the corresponding minimum parallel execution time of running proposed techniques with several parallel processes on the described system using test input images.

All recorded parallel execution time values demonstrate a significant execution time reduction compared with the serial execution time values for the tested techniques regardless the percent of reduction. This implies that all the proposed techniques had achieved execution acceleration compared with the sequential implementation.

Table 2: Execution time reduction experimental results

Image, Serial Execution Time	Parallel Techniques Minimum Parallel Execution time (Sec.)			
	Technique1, NASHT1		Technique2, NASHT2	
	Send	Bcast	Send	Bcast
Brain, 5.26 Sec.	3.81	4.02	1.22	2.07
Lena, 46.90 Sec.	18.49	19.25	13.72	19.84
Picnic, 215.64 Sec.	73.96	75.55	76.88	133.95

Depending only on comparing parallel execution time with the serial one is not an efficient mechanism to study the behavior of the tested technique against specific parameters such as number of running parallel processes and image size.

Experiment 3 was carried out to study the effect of the number of running parallel process and image size on the relative speedup of the tested technique. Relative speedup is computed by the formula,  $Speedup = T_S / T_P$ , where  $T_S$  is the execution time of sequential technique and  $T_P$  is the execution time of corresponding parallel one. This experiments has two folds, the first one is studying the effect of the number of running parallel processes used in the technique execution on the execution behavior.

The second one is to study how the image size affects the performance of tested technique. In this experiment, the sequential versions that correspond to each parallel technique are executed with an input image and the execution time is then recorded for each version.

Each parallel technique is repeatedly executed with the same input image increasing the number of parallel processes in each run; the execution time and relative speedup of each run are then computed and recorded for each version.

The same scenario is followed with other images having different sizes to observe the effect of image size on execution behavior.

Table 3 shows the results of implementing this experiment. For readability, "npr", "ex" and "s" denote the number of parallel processes, parallel execution time and relative speedup respectively.

The results show that relative speedup of both versions of Technique1, in general, decreases as the number of processes increases, since all processes send their generated layers to root process to be written which leads to extra overhead on master process to finalize its amount of work.

Table 3: Number of processes and Image size experimental results

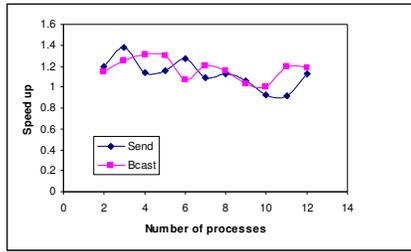
Technique/ Version		Technique1, NASHT1 Send			Technique1, NASHT1 Bcast			Technique2, NASHT2 Send			Technique2, NASHT1 Bcast	
npr		ex	s	ex	s	npr	ex	s	ex	s		
Brain Image	2	4.39	1.20	4.58	1.15	3	3.26	1.61	4.32	1.22		
	3	3.81	1.38	4.18	1.26	5	2.40	2.20	3.75	1.40		
	4	4.65	1.13	4.02	1.31	7	2.10	2.50	3.12	1.69		
	5	4.57	1.15	4.04	1.30	9	1.80	2.94	2.56	2.06		
	6	4.13	1.27	4.91	1.07	11	1.68	3.13	2.82	1.87		
	7	4.85	1.09	4.35	1.21	13	1.22	4.30	2.08	2.54		
	8	4.67	1.13	4.55	1.16							
	9	4.96	1.06	5.08	1.04							
	10	5.70	0.92	5.25	1.00							
	11	5.75	0.91	4.40	1.20							
	12	4.69	1.12	4.44	1.19							
	Technique/ Version		Technique1, NASHT1 Send			Technique1, NASHT1 Bcast			Technique2, NASHT2 Send			Technique2, NASHT2 Bcast
npr		ex	s	ex	s	npr	ex	s	ex	s		
Lena Image	2	22.01	2.13	22.78	2.06	3	23.77	1.97	41.93	1.12		
	3	24.60	1.91	23.96	1.96	5	19.12	2.45	36.50	1.28		
	4	21.50	2.18	20.80	2.26	7	17.89	2.62	37.08	1.27		
	5	19.71	2.37	21.26	2.21	9	15.65	3.00	34.87	1.35		
	6	19.37	2.42	19.25	2.44	11	14.26	3.29	26.55	1.77		
	7	18.49	2.54	20.12	2.33	13	13.72	3.42	19.84	2.36		
	8	21.06	2.23	24.40	1.92							
	9	20.31	2.31	20.57	2.28							
	10	25.53	1.84	23.03	2.04							
	11	28.00	1.68	27.47	1.71							
	12	30.51	1.54	25.05	1.87							
	Technique/ Version		Technique1, NASHT1 Send			Technique1, NASHT1 Bcast			Technique2, NASHT2 Send			Technique2, NASHT2 Bcast
npr		ex	s	ex	s	npr	ex	s	ex	s		
Picnic Image	2	104.75	2.06	104.32	2.07	3	110.5	1.95	194.3	1.11		
	3	109.53	1.97	109.87	1.96	5	88.49	2.44	170.7	1.26		
	4	83.08	2.60	85.45	2.52	7	90.20	2.39	169.2	1.27		
	5	91.69	2.35	90.72	2.38	9	90.21	2.39	155.6	1.39		
	6	75.18	2.87	75.55	2.85	11	76.88	2.80	133.9	1.61		
	7	73.96	2.92	79.14	2.72	13	97.87	2.20	127.2	1.70		
	8	83.69	2.58	84.04	2.57							
	9	77.12	2.80	78.28	2.75							
	10	104.77	2.06	103.16	2.09							
	11	105.58	2.04	116.84	1.85							
	12	95.97	2.25	97.80	2.20							

In contrast, relative speedup of both versions of Technique1 increases as the image size increases due to decreasing the computation/ communication ratio for larger images. Although the performance of "Send" version is very close to that of "Bcast" version, the first version gained a slight higher speedup value than the second one. These observations can be noticed in as shown in Figure 16-a, Figure 17-a, and Figure 18- a.

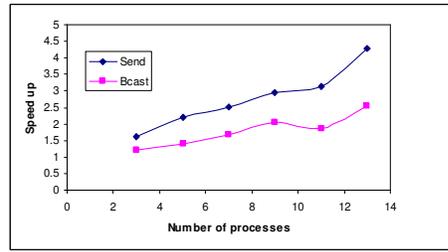
Concerning Technique2, relative speedup of both versions of Technique2 increases as the number of process increases, since odd ranked processes send their generated layers to the neighbour processes to be written which leads to less overhead on even ranked processes to finalize their amount of work, this is obvious in Figure 16-b, Figure 17-b, and Figure 18-b.

Relative speedup of both versions of Technique2 decreases as the image size increases but still increases as the number of processes increases.

It is also noticed that "Send" version performs better than "Bcast" version especially for small sized images except in case of larger number of processes as shown in Figure 18-b.

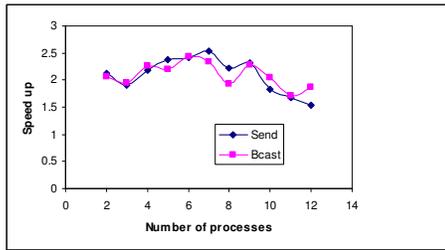


(a) NASHT1 : Version 1, Version 2

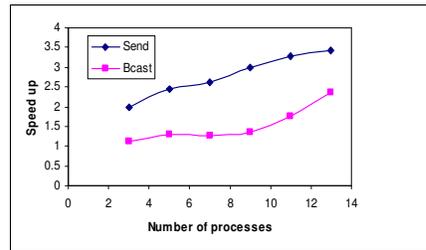


(b) NASHT 2: Version 1, Version 2

Figure 16. Brain 165x 158 – Serial: 5.26 Sec. – Max. Speedup : 1.38 – 1.31 , 4.29 -2.53

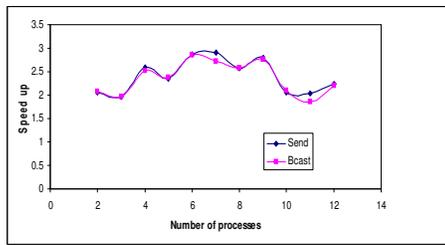


(a) NASHT 1: Version 1, Version 2

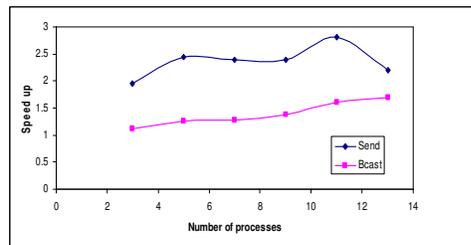


(b) NASHT 2: Version 1, Version 2

Figure 17. Lena 512 x 512 – Serial: 46.90 Sec. – Max. Speedup : 2.54 – 2.44, 3.42 – 2.36



(a) NASHT 1: Version 1, Version 2



(b) NASHT 2: Version 1, Version 2

Figure 18. Picnic 1280 x 960 – Serial: 215.64 Sec – Max. Speedup : 2.92 – 2.85, 2.80 – 1.70

From the above discussion, we can summarize the results as shown in Table 4 that guides to recommend using of Technique2-"Send" version in case of small and medium sized images.

Table 4: Image size – Relative speedup summary

Image, Rows x Columns	Parallel Technique , Version Maximum Relative Speedup			
	NASHT1		NASHT2	
	Send	Bcast	Send	Bcast
Brain, 165x 158	1.38	1.31	4.29	2.53
Lena, 512x 512	2.54	2.44	3.42	2.36
Picnic, 1280 x 960	2.92	2.85	2.80	1.70

Using of Technique1-"Send" version is recommended in case of large sized images. This does not mean that the other versions of both techniques are not applicable or not recommended for use since all of them generate valid image layers and gains notable relative speedup.

## 6. CONCLUSION AND FUTURE WORK

We introduced two parallel techniques not only to apply edge detection to generate a set of layers for an input image but also to study the effect of the number of parallel running processes and image size on the performance of the proposed techniques.

Our techniques can generate an arbitrary set of layers in a single parallel run instead of generating a unique layer as in traditional case; this helps in selecting the layers with high quality edges among the generated ones. Each presented parallel technique has two versions based on point to point communication "Send" and collective communication "Bcast" functions. All of the techniques gained higher speedup than that of sequential ones.

The effect of the number of running parallel processes and image size on the performance of the proposed techniques was analyzed. In future, we plan to modify some communication patterns in the proposed techniques to gain higher speedup. Extending the used hardware platform and studying the effect of larger images are also to be examined.

## REFERENCES

- [1] Poonam, S. Deokar, (2015) "Implementation of Canny Edge Detector Algorithm using FPGA" , IJSET - International Journal of Innovative Science, Engineering & Technology, Vol. 2 No. 6, pp 112 - 115.
- [2] Gholamali Rezai-Rad & Majid Aghababaie, (2006) "Comparison of SUSAN and Sobel Edge Detection in MRI Images for Feature Extraction", IEEE transaction on Information and Communication Technologies, Vol. 1, pp 1103 - 1107.
- [3] Raman Maini & Dr. Himanshu Aggarwal, (2009) "Study and Comparison of Various Image Edge Detection Techniques", International Journal of Image Processing (IJIP), Vol. 3, No. 1, pp 1-12.
- [4] L.S. Davis, (1975) "A survey of edge detection techniques", Computer Graphics and Image Processing, Vol. 4, No. 3, pp 248-260.
- [5] Tarek A. Mahmoud & Stephen Marshall, (2008) "Medical Image Enhancement Using Threshold Decomposition Driven Adaptive Morphological Filter", 16th European Signal Processing Conference (EUSIPCO 2008), EURASIP.
- [6] P.Subashini & M.Krishnaveni & Suresh Kumar Thakur, (2010) "Quantitative Performance Evaluation on Segmentation Methods for SAR ship images", Proceedings of the Third Annual ACM Bangalore Conference,.
- [7] Saurabh Singh & Dr. Ashutosh Datar, (2013) "EDGE Detection Techniques Using Hough Transform", International Journal of Emerging Technology and Advanced Engineering, Vol. 3, NO. 6, pp 333 - 337.
- [8] Salem Saleh Al-amri & N.V. Kalyankar & Khamitkar S.D, (2010) " Image Segmentation by Using Thershod Techniques", Journal of Computing, Vol. 2, No. 5, MAY, pp 83 - 86.

- [9] Alexander Drobchenko & Jarkko Vartiainen & Joni-Kristian Kämäräinen & Lasse Lensu & Heikki Kälviäinen, (2011) “ Thresholding Based Detection of Fine and Sparse Details“, *Frontiers of Electrical and Engineering, China*, Vol. 6, No. 2, , pp 328 - 338.
- [10] Derek Bradley & Gerhard Roth, (2007) “Adaptive Thresholding using the Integral Image“, *Journal of Graphics, GPU, and Game Tools*, Vol. 12, No. 2, pp 13-21.
- [11] Simranjit Singh Walia & Gagandeep Singh, (2014) “Color based Edge detection techniques– A review “, *International Journal of Engineering and Innovative Technology (IJEIT)*, Vol. 3, No. 9, March, pp 297-301.
- [12] S. Jansi & P. Subashini, (2012) “Optimized Adaptive Thresholding based Edge Detection Method for MRI Brain Images“, *International Journal of Computer Applications*, Vol. 51, No.20, pp 1-8.
- [13] Li, J. & Ding, S., (2011) “A research on improved canny edge detection algorithm“, in *International Conference on Applied Informatics and Communication*, pp 102-108.
- [14] Ghassan F. Issa & Hussein Al-Bahadili & Shakir M. Hussain, (2012) “ Development and Performance Evaluation of A Lan-Based Edge-Detection Tool“, *International Journal on Soft Computing ( IJSC )* Vol.3, No.1, pp 121-135.
- [15] Mohd Azam Osman & Muqhtar Yassin Mohamad & Rosni Abdullah, (2008)“ Parallelizing an Edge Detection Algorithm for Image Recognition to Classify Paddy and Weeds Leaf on Sun Fire Cluster System“, *7th WSEAS Int. Conf. on Software Engineering, Parallel and Distributed Systems (SEPADS '08)*, University of Cambridge, UK, pp 56-60.
- [16] Chandrashekar N.S. & Dr. K.R. Nataraj, (2012) “A Distributed Canny Edge Detector and Its Implementation on FPGA“, *International Journal Of Computational Engineering Research (ijceronline.com)* Vol. 2, No. 7, pp 177-181.
- [17] A. Z. Brethorst & N. Desai, D. P. Enright & R. Scrofano, (2011) “Performance evaluation of Canny edge detection on a tiled multicore architecture.“ in *Society of Photo-Optical Instrumentation Engineers (SPIE) ConferenceSeries*, Vol. 7872, pp. 1–8.
- [18] Taieb Lamine Ben Cheikh & Giovanni Beltrame & Gabriela Nicolescu & Farida Cheriet & Sofi`ene Tahar,( 2012) “Parallelization Strategies of the Canny Edge Detector for Multi-core CPUs and Many-core GPUs“, *Conference Proc. of the 10th IEEE Intl. NEWCAS Conference*, pp 49-52.
- [19] Christos Gentsos & Calliope- LouisaSotirpoulou & Spiridon Nikolaidis Nikolaos Vassiliadis (2010) “Real-Time Edge Detection Parallel Implementation for FPGAs“, *17th IEEE International Conference on Electronics, Circuits, and Systems, ICECS 2010, Athens, Greece*, pp 499-502.
- [20] Jones, Donald R. & Elizabeth R. Jurrus & Brian D. Moon & Kenneth A. Perrine, (2003)“Gigapixel-size Real-time Interactive Image Processing with Parallel Computers,“ *Parallel and Distributed Processing Symposium*.
- [21] Aaron Hagan & Ye Zhao, (2009)“ Parallel 3D Image Segmentation of Large Data Sets on a GPU Cluster“, *5th International Symposium, ISVC 2009, Las Vegas, NV, USA*, pp. 960–969
- [22] Luis H.A. Lourenc,o & Daniel Weingaertner & Eduardo Todt,(2012)“ Efficient implementation of Canny Edge Detection Filter for ITK using CUDA“, *13th Symposium on Computer Systems*, pp 960–969.
- [23] Prakash K. Aithal & U. Dinesh Acharya & Rajesh Gopakumar, (2015)“Detecting the Edge of Multiple Images in Parallel“, *International Journal of Computer, Electrical, Automation, Control and Information Engineering* Vol.9, No.7, pp 1442–1445.
- [24] Juan C. Pichel & David E. Singh & Francisco F. Rivera, (2007)“ A Parallel Framework for Image Segmentation Using Region Based Techniques“, *Vision Systems: Segmentation and Pattern Recognition, I-Tech, Vienna, Austria*, pp 81-98.
- [25] J. Gao & N. Liu, (2012)“An Improved Adaptive Threshold Canny Edge Detection Algorithm “, in *Computer Science and Electronics Engineering (ICCSEE)*, *International Conference on*, pp. 164-168.
- [26] J. F. Canny, (1986) “A computational approach to edge detection“, *IEEE Trans. Pattern Analysis and Machine Intelligence*, Vol., No. 6, pp. 679-698.
- [27] Samir Kumar Bandyopadhyay, (2011)“Edge Detection in Brain Images“, *International Journal of Computer Science and Information Technologies*, Vol. 2, No.2, pp 884-887.
- [28] R.A. Haddad & A.N. Akansu, (1991) “A Class of Fast Gaussian Binomial Filters for Speech and Image Processing,“ *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. 39, March, pp 723-727.
- [29] W. Gropp, (2002) “MPICH2: A New Start for MPI Implementations“, In *Recent Advances in PVM and MPI: 9th European PVM/MPI Users’ Group Meeting*, Linz, Austria.

- [30] Bernd Mohr & Jesper Larsson Träff & Joachim Worringer & Jack Dongarra, (2006)“Recent Advances in Parallel Virtual Machine and Message Passing Interface“, 13th European PVM/MPI User's Group Meeting Proceedings, Bonn, Germany, pp 17-20.
- [31] Alaa Ismail El-Nashar, (2011) “Performance of MPI Sorting Algorithms on Dual-core Processor Windows-based Systems“, International Journal of Distributed and Parallel Systems (IJDPS) Vol.2, No.3, pp 1–14.
- [32] A. Chan D. Ashton & R. Lusk, and W. Gropp, (2007)“Jumpshot-4 Users Guide“ Mathematics and Computer Science Division, Argonne National Laboratory.
- [33] Omer Zaki, Ewing Lusk & William Gropp & Deborah Swider, (1999) “Toward Scalable Performance Visualization with Jumpshot“, International Journal of High Performance Computing Applications, Vol. 13, No. 3 pp 277 – 288.

## **AUTHOR**

**Alaa I. Elnashar** was born in Minia, Egypt, on November 5, 1967. He received his B.Sc. and M.Sc. from Faculty of Science, Department of Mathematics (Math. & Comp. Science), and Ph.D. from Faculty of Science, Department of Computer Science, Minia University, Egypt, in 1988, 1994 and 2005. He is an associate professor in Faculty of Science, Computer Science Dept., Minia University, Egypt.



Dr. Elnashar was a postdoctoral fellow at Kanazawa University, Japan. His research interests are in the area of Software Engineering, Software Testing, and parallel programming.

Now, Dr. Elnashar is an associate professor, Department of Information Technology, College of Computers and Information Technology, Taif University, Saudi Arabia