

1990

Adaptive Histogram Equalization, a Parallel Implementation

Charles W. Kurak Jr.
University of North Florida

Suggested Citation

Kurak, Charles W. Jr., "Adaptive Histogram Equalization, a Parallel Implementation" (1990). *UNF Graduate Theses and Dissertations*. 260.
<https://digitalcommons.unf.edu/etd/260>

This Master's Thesis is brought to you for free and open access by the Student Scholarship at UNF Digital Commons. It has been accepted for inclusion in UNF Graduate Theses and Dissertations by an authorized administrator of UNF Digital Commons. For more information, please contact [Digital Projects](#).

© 1990 All Rights Reserved

ADAPTIVE HISTOGRAM EQUALIZATION
A PARALLEL IMPLEMENTATION

by

Charles W. Kurak Jr.

A thesis submitted to the
College of Computer and Information Sciences
in partial fulfillment of the requirements for the degree of

Master of Science in Computer and Information Sciences

UNIVERSITY OF NORTH FLORIDA
COLLEGE OF COMPUTER AND INFORMATION SCIENCES

December, 1990

The thesis "Adaptive Histogram Equalization: A Parallel Implementation" submitted by Charles W. Kurak Jr. in partial fulfillment of the requirements for the degree of Master of Science in Computer and Information Sciences has been

Approved by the thesis committee:

Signature Deleted

Date

12/10/90

Thesis Adviser and Committee Chairman

Signature Deleted

12/10/90

Signature Deleted

12/10/90

Accepted for the College of Computer and Information Sciences:

Signature Deleted

12/10/90

Interim Dean

Accepted for the University:

Signature Deleted

12-12-90

Vice-President for Academic Affairs

ACKNOWLEDGEMENTS

I wish to thank my advisor and thesis committee chairman, Dr. Yap S. Chua, for his continuous support, advice, and encouragement during this work. The members of my committee also deserve credit for their support. Dr. Ralph M. Butler was extremely helpful with parallel algorithm design and by supplying the parallel programming package. Dr. Charles N. Winton, as always, continued to offer encouragement and material support throughout the project.

I wish to acknowledge the support I have received from members of the University of North Carolina at Chapel Hill community. Dr. Stephen M. Pizer introduced me to Adaptive Histogram Equalization and has been supportive of my efforts with his time and suggestions. Dr. James M. Coggins provided continual support during the project. Dr. Graham Gash aided with the 'usr/image' software developed at UNC-CH. This package was one of the tools utilized for data conversion and interactive image display. Mr. Robert Cromartie aided in the verification of the border mirroring methodology and the image results in general. The UNC Medical Image Display Research Group provided the medical images.

Saving the most important for last, my ever-supportive and cherished wife, Beverly, helped me through the long hours and seemingly endless nights and weekends. Her background as a businesswoman, an educator, and a loyal wife and mother continues to be an asset to my efforts.

CONTENTS

List of Figures	vi
Abstract	vii
Chapter 1: Introduction	1
1.1 Digital Image Processing Historical Overview	1
1.2 Histogram Equalization	2
1.3 Adaptive Histogram Equalization (AHE)	3
1.4 AHE Limitations	6
1.4.1 Time Performance	7
1.4.2 Edge Considerations	8
1.4.3 Excessive Computations	8
1.5 Parallel Processing	9
1.5.1 Parallel Programming Paradigms	10
1.5.2 Message Passing Paradigm	11
1.5.3 Model Selection	12
Chapter 2: Hypothesis	13
Chapter 3: Implementation	14
3.1 C language and Unix Operating System	14
3.2 Message Passing Paradigm	14
3.3 Multiple image copies	15
3.4 Image Border	16
3.5 Value Look-up Table	20
3.6 Coordination between Master and Slaves	22

Chapter 4: Tools	
4.1 DISPLAY for the PC	26
4.2 'disp' for the Sun workstation with X Windows	27
4.3 Image and Histogram Production for Postscript	27
4.4 'usr/image' from UNC-CH	28
4.5 Thesis publication software	28
Chapter 5: Testing & Results	29
Chapter 6: Discussion of Results	30
Chapter 7: Conclusion	38
Chapter 8: Future Work	39
8.1 SIMD architecture	39
8.2 Interactive viewing of image during processing	40
8.3 The AHE Family of Algorithms	41
8.4 Three-dimensional AHE	41
References	43
Appendix A: Source code for Parallel AHE (master process)	45
Appendix B: Source code for Parallel AHE (slave process)	54
Appendix C: Test Results	60
Vita	64

FIGURES

Figure 1: MRI Scan - Original Image	4
Figure 2: MRI Scan - after Contrast Stretching	4
Figure 3: Brain - Original Image	5
Figure 4: Brain - after Contrast Stretching	5
Figure 5: The AHE Algorithm	6
Figure 6: Number of required comparisons	7
Figure 7: One possible method of handling borders	17
Figure 8: Chest - after AHE with contextual region of 16 x 16	19
Figure 9: Chest - after AHE with contextual region of 32 x 32	19
Figure 10: Mirroring across the image boundaries	20
Figure 11: Message Passing Model	23
Figure 12: Master Process - Pseudo-Code	25
Figure 13: Slave Process - Pseudo-Code	25
Figure 14: Chest - Original Image	31
Figure 15: Chest - after AHE with contextual region of 64 x 64	31
Figure 16: Brain - Original Image	32
Figure 17: Brain - after AHE with contextual region of 64 x 64	32
Figure 18: AHE Results - Contextual Region of 8 x 8	33
Figure 19: AHE Results - Contextual Region of 16 x 16	33
Figure 20: AHE Results - Contextual Region of 32 x 32	34
Figure 21: AHE Results - Contextual Region of 64 x 64	34
Figure 22: AHE Results - 26-node machine	37
Figure 23: Dividing the image into interlaced sub-problems	41

ABSTRACT

Adaptive Histogram Equalization (AHE) has been recognized as a valid method of contrast enhancement. The main advantage of AHE is that it can provide better contrast in local areas than that achievable utilizing traditional histogram equalization methods. Whereas traditional methods consider the entire image, AHE utilizes a local contextual region.

However, AHE is computationally expensive, and therefore time-consuming. In this work two areas of computer science, image processing and parallel processing, are combined to produce an efficient algorithm. In particular, the AHE algorithm is implemented with a Multiple-Instruction-Multiple-Data (MIMD) parallel architecture. It is proposed that, as MIMD machines become more powerful and prevalent, this methodology can be applied to not only this particular algorithm, but also to many others in its class.

Chapter 1

INTRODUCTION

Digital image processing has aided in the enhancement and restoration of images for over a quarter of a century. This work focuses on one particular algorithm among the many available. In particular, it will be proposed that the algorithm in question can be improved with the application of knowledge from a separate branch of computer science. To enable the reader to better understand the algorithm, relevant background information will be presented.

1.1 Digital Image Processing Historical Overview

Digital image processing techniques had their significant inception in the 1960s with the space program. The requirement to enhance the quality of the photographs returned by the early space probes motivated research in this area. Originally this technology was applied to space imagery, however it was soon realized that other areas could also benefit. Consequently, the needs of the medical field attracted the attention of researchers. The methods utilized by the space industry were applied. With the advent of the X-ray and later Nuclear Magnetic Resonance Imaging (NMRI or MRI), Positron Emission Tomography (PET Scans), Computerized Assisted Tomography (CAT Scans), and Ultra-sound Imaging, the quantity of medical data increased dramatically. Simultaneously researchers sought better methods to enhance these images.

The test data utilized for this work originated from three different types of data collection methods. The chest image is one slice of a CAT scan (see Figure 14 on page 31). The left facing head is slice number 50 of a 109-slice MRI scan (see Figure 1 on page 4). The right facing head labeled 'brain' is a portal image taken during Radiotherapy Treatment (see Figure 16 on page 32). Each file arrives for enhancement in a raw image format. In raw image format the intensity values are contained in a file in row major order, with multiple planes being stored consecutively from front to back.

1.2 Histogram Equalization

One of the most basic and simple, yet powerful tools in image enhancement is the histogram. This tool is simply a frequency count of the intensity levels of each digitized point, or pixel, contained in the image. Utilizing the information contained in a histogram allows us to improve the contrast of an image. This information may be hidden from the human eye; however it is readily acquired by use of a computer. Whereas the Human Visual System (HVS) can only distinguish approximately 100 levels of gray shades, the computer can detect an almost infinite number of levels. The practical limiting factor for the computer is the number of various intensity levels recognizable by the digitizing equipment.

For example, using the histogram, under-developed or over-developed photographs can be restored or enhanced to produce an image usable by the HVS. Assuming the histogram reveals a number of intensity levels all located in the low intensity range, each current value can be mapped to a new level so that the new histogram

is scaled to cover the entire range of available intensity levels. Note the images and their accompanying histograms in Figures 1 and 2 on page 4.

1.3 Adaptive Histogram Equalization

The Histogram Equalization method of contrast enhancement functions extremely well for images that are underexposed or overexposed, i.e. images with very little overall contrast. However, there exist images whose histograms cover the entire spectrum of intensity values but reveal little contrast in localized areas (see Figures 3 and 4 on page 5). In this case a variation of histogram equalization can be applied. Developed independently by Hummel [Hummel75, Hummel77], Ketcham [Ketcham76], and Pizer [Pizer81a, Pizer81b], Adaptive Histogram Equalization (AHE) has been successfully applied to images obtained from numerous sources. Although this thesis concentrates on medical imagery, the success of the method used depends on the characteristics of the image, not the content. The identical methodology can be applied to other areas of interest.

The technique is quite simple. The algorithm is included in Figure 5 on page 6. Each pixel is ranked by its intensity level as compared to its neighboring pixels' intensity values. The pixel is then assigned a new value in the available intensity range proportionate to its rank. For example, if a pixel's rank is #8 of 64 and the available intensity range of the display device is 0-255, its new value would be one-eighth of 255 or 32. This new value is assigned to a second image (an output image) so as to not disturb the original ranking of each of the pixels.

MRI Scan - Original Image.

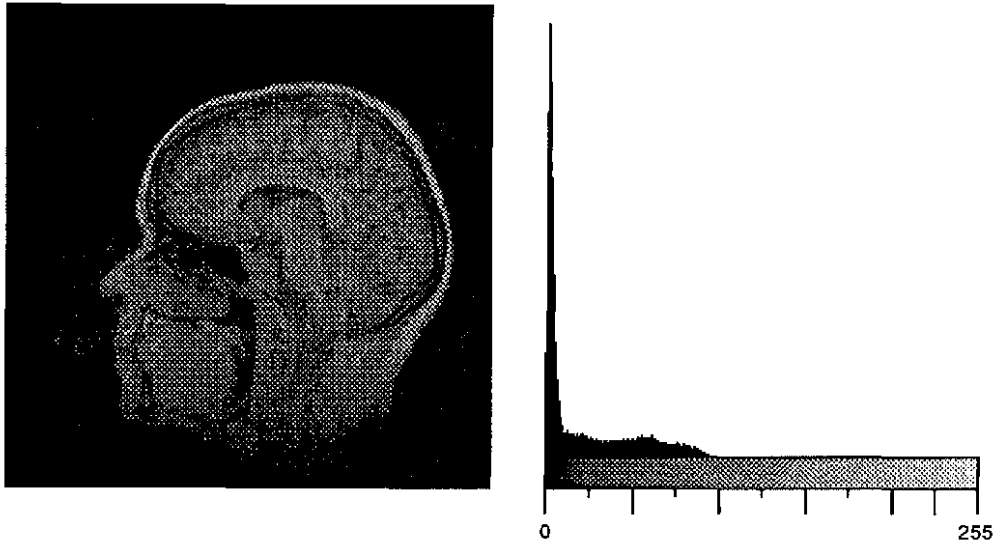


Figure 1: MRI Scan - Original Image.

MRI Scan - after Contrast Stretching.

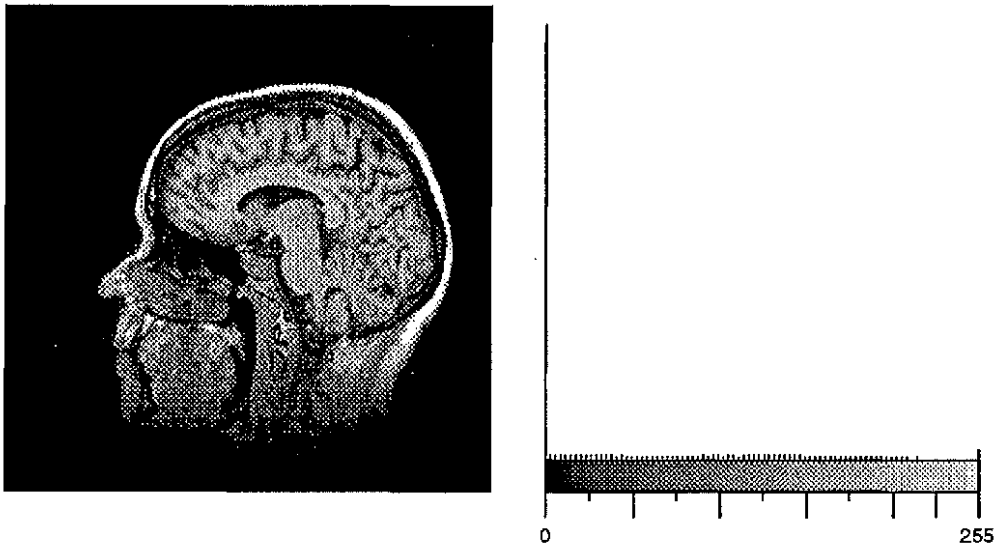


Figure 2: MRI Scan - after Contrast Stretching.

Brain: original image.

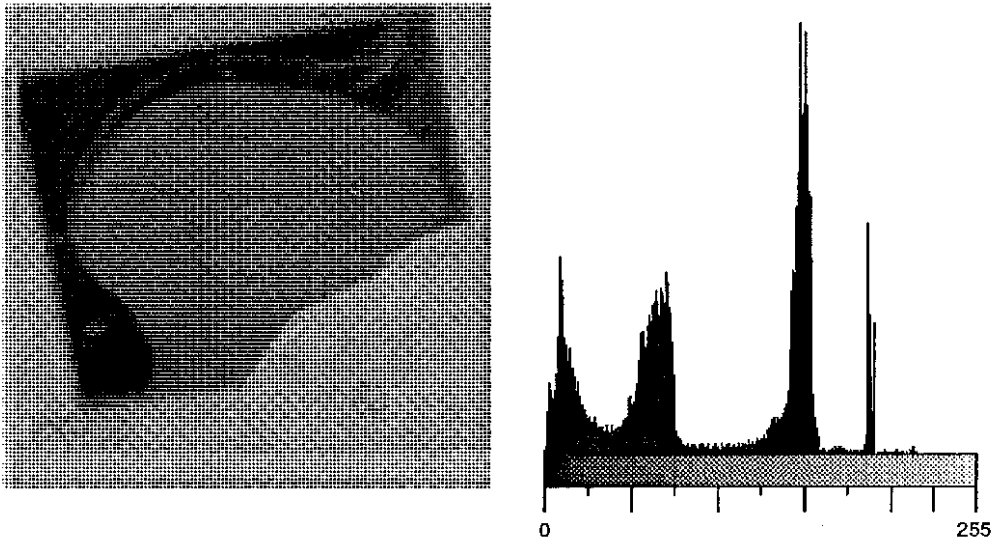


Figure 3: Brain - Original Image.

Brain: after contrast stretching.

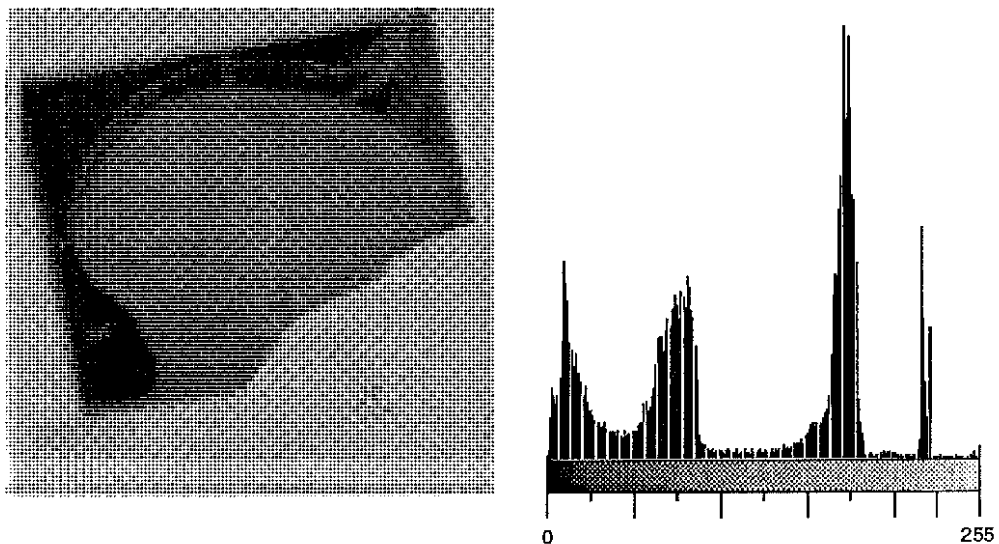


Figure 4: Brain - after Contrast Stretching.

AHE Algorithm

```
for each (x,y) in image do
{
    rank = 0
    for each (i,j) in contextual region of (x,y) do
    {
        if image[x,y] > image[i,j] then
            rank = rank + 1
        }
    output[x,y] = rank * max_intensity / (# of pixels in contextual region)
}
```

Figure 5: The AHE Algorithm.

This method enhances the contrast based on the local area or contextual region rather than the entire image. AHE is currently being utilized in both research at the University of North Carolina at Chapel Hill, and in production systems such as Mayo Clinic's "Analyze" software.

1.4 AHE Limitations

While Adaptive Histogram has the advantage of being able to enhance contrast in local areas, there is a price to pay. Three areas of concern surfaced during the initial investigation: time, border considerations, and excessive computations. These are discussed below.

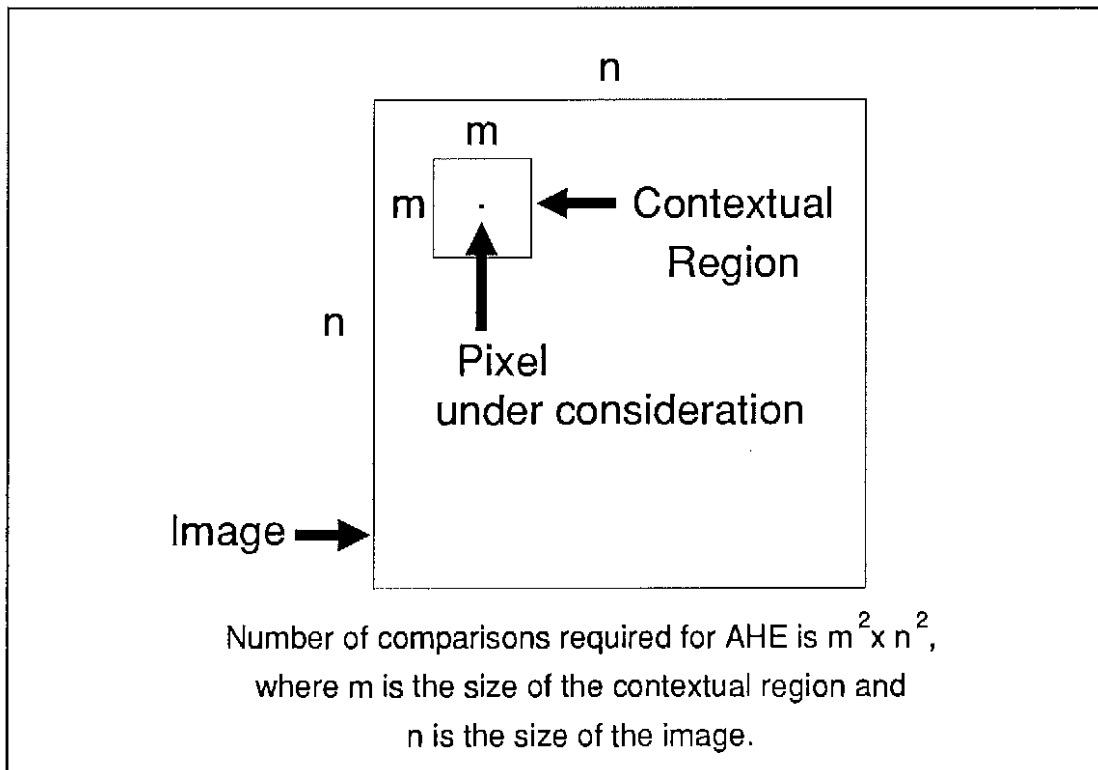


Figure 6: Number of required comparisons.

1.4.1 Time Performance

Unfortunately, applying AHE to an image is computationally expensive. Each image requires $m^2 \times n^2$ comparisons where m is the size of the contextual region and n is the size of the image. (See Figure 6 above.) For this to be a truly valid tool for a radiologist or a medical doctor, the processing must be completed in a reasonable amount of time. Initial studies by the author determined that in excess of one hour on a 4 MIPS Sequent Symmetry machine was required to process a 512 x 512 8-bit image with a 64 x 64 contextual region. Consultation with Dr. Pizer and Mr. Robert Cromartie at UNC-CH indicated that lowering the processing times would be extremely beneficial.

Three approaches were considered. First, simply wait until faster hardware is available. Second, one could build specialized hardware. MAHEM [Austin87] is one such research effort that utilizes a parallel processing architecture. Third, search for an alternative. In this work the author proposes one alternative: parallelism on commercially available hardware. The majority of the remainder of this work addresses this approach.

1.4.2 Border Considerations

An issue that surfaced during the investigation of the AHE algorithm is that of handling pixels whose contextual regions overlapped the borders. Current literature [Ericksen90] suggests that for convenience, the contextual region indices simply utilize a wrap-around technique. As suggested by the author to Dr. Pizer during a personal meeting in May 1990, this technique is invalid. For example, if the area near the border under consideration is relatively dark (low intensity values are present), and the values on the opposite edge are relatively light (high intensity values are present), the resulting rank calculation would be unsatisfactorily biased.

A mirroring technique has been suggested by the author. This technique has been implemented in this work and is explained in Section 3.4.

1.4.3 Excessive Computations

In addition to the number of comparisons required by the AHE algorithm, an additional issue surfaced during the implementation. This is the number of

excessive computations required to calculate each new intensity value. In the AHE algorithm in Figure 5, on page 6, note the calculation for the new intensity value, $output[x,y]$. This requires one multiplication operation and one division operation for each pixel. For example, in a 256×256 image there would be 65,536 (256^2) integer multiplications and 65,536 (256^2) integer divisions required.

One method of reducing the number of calculations is to precalculate the $max_intensity / (\# \text{ of pixels in contextual region})$. This eliminates an integer division calculation, however it replaces the integer multiplication with a floating point multiplication, which may be more time consuming.

Another method, which reduces the total number of calculations to the number of pixels in the contextual region, was devised utilizing a look-up table. The method is discussed in Section 3.5.

1.5 Parallel Processing

The concept of accomplishing more work in less time by employing additional resources is not original by any consideration. After studying the AHE algorithm it can be determined that one can take advantage of the positive aspects of parallel processing. The following is a brief discussion of the programming models available, the one selected for the implementation, and reasons for this choice.

1.5.1 Parallel Processing Programming Paradigms

There are three well known programming models for parallel processing in an MIMD environment. These are the Askfor-Monitor, Message Passing, and Communicating Clusters [Boyle87]. The Askfor-Monitor requires a shared memory machine. That is, each processor must have access to a common memory area that is shared by all of the processors. The processes communicate through this common area. The monitor is a data structure that controls the activities of the processes.

Message Passing can operate either across a network or on a shared memory machine. The processes communicate by passing messages. One process is designated as the master and controls the activities of the other processes, also known as slaves. The ability to operate across a network allows the use of non-multi-processing hardware to function together in a multi-processing fashion.

The third model is Communicating Clusters. Essentially this is a combination of Askfor Monitors and Message Passing. A master process on one machine invokes processes on other machines that have shared memories. One process on each of these machines is designated by the master to be a cluster master. They in turn designate the other processes on their machine to be local slaves. Each cluster coordinates its work via an Askfor-Monitor in shared memory. The master coordinates the work among the clusters via Message Passing. Thus, the Communicating Clusters model can be custom-built depending on the available hardware.

1.5.2 Message Passing Paradigm

The model selected for this work was Message Passing. This decision was based on portability across several different architectures. A brief example is presented here to acquaint the reader with the Message Passing model.

Suppose we were given the task of finding all of the prime numbers within a given range. With a large enough range this becomes non-trivial. Also, suppose we were given multiple processors whose processes could communicate via messages. We could then take advantage of a parallel algorithm for solving the problem.

One process is designated as the initiating process (master). Its function is to read the problem, decompose the problem into sub-problems, activate processes on other processors (slaves), hand out the work, collect the answers, and report a solution. The remaining processes perform the actual work, under the control of the master.

The master is the first to be invoked. It in turn invokes the proper number of slave processes as indicated in a user-defined parameter setup file. The master then divides the work into a series of sub-problems and places a representation for each in a problem queue for dispatching to the slaves. As each slave is initiated, it sends a message to the master requesting work. The master places a representation for each slave in a slave queue indicating that it is available to perform work. The master then proceeds to the main control loop of the program. If the problem queue is not empty, the master in turn dispatches the sub-problems to the

appropriate slaves. Each slave computes its sub-problem, sends its solution back to the master, and requests additional work. When the problem queue is empty and the slave queue contains all of the invoked slaves, the problem is declared complete, and the master leaves the main control loop. The master sends a control message to each slave to terminate. The complete solution is reported to the user, normally as an output file. The master then terminates.

1.5.3 Model Selection

The Message Passing model was selected for this work due to its ability to operate both in a shared memory environment and across a network. The parallel programming package utilized for the implementation was designed for a shared memory machine. However, there is a compatible package that can be substituted without any requirement for source code modification. The desire to utilize this code as an ongoing research tool influenced the selection of the most portable model.

Chapter 2

HYPOTHESIS

There are parallel processing techniques which will substantially improve the time performance of Adaptive Histogram Equalization.

Given the large amount of similar computation necessary to complete the AHE algorithm, the premise is that applying parallel processing techniques will greatly reduce the amount of overall time required to process an image. Assuming that one image requires amount k time, then dividing the work among n processors would yield an ideal completion time of k/n . Another way of saying this, is that one could obtain a speed-up of factor n .

The ideal speed-up of factor n is not realistic. There is necessary overhead relating to the coordination of the n processes, problem distribution, and solution reporting. The remaining question is then, how closely can one approach the ideal?

Chapter 3

IMPLEMENTATION

3.1 'C' Language and Unix Operating System

The programming language 'C' was selected for this implementation. Other languages could have been utilized. However the parallel programming package supplied by Dr. Ralph Butler [Butler90] is implemented in the language 'C'. For reasons of ease and compatibility the choice was obvious. Also, future work interfacing this software with the 'X Windows' software system will need to be in 'C'. The Unix operating system was selected, as Dr. Butler's package is implemented to run under Unix.

3.2 Message Passing Paradigm

This particular model was selected from [Boyle87] mainly due to its portability. As mentioned in 1.5.1, Message Passing was selected for its portability across a variety of platforms. The source code for this work is included in Appendices A & B. The majority of the code is related to the coordination of the work between the master and the slaves.

The master's responsibilities are to determine the problem and number of slaves to be used, invoke the slaves, coordinate the work, gather the solution, terminate the

slaves' existence, and output the solution. The master source code is included in `pmaster.c` in Appendix A.

Each slave's responsibility is to load the image into its local memory, compute sections of the image as requested by the master, and pass the completed problem solutions to the master. The slave source code is included in `pslave.c` in Appendix B.

3.3 Multiple Image Copies

During the design phase it was determined that passing a problem from the master to a slave can be handled in three different methods. First the master can send a copy of the sub-image to be processed to a slave. Given a sub-problem size of 64 x 64, this would be 4096 bytes. Also, given that the algorithm requires the values of pixels in neighboring areas, and given a typical 64 x 64 contextual region size, the overall sub-image size that needs to be transmitted is 128 x 128 or 16,384 bytes. This would have to be sent for each sub-problem. Given a 256 x 256 image, and a sub-problem size of 64 x 64, this would yield 16 sub-problems. Thus, a total of 262,144 bytes (16,384 x 16) would need to be transmitted.

A second method would be to send the entire image at once. Then only a representation of each sub-problem would need to be transmitted for each piece of work. This would involve transmitting 65,536 bytes. This is still a large amount of overhead.

A third method, where only the file name be sent once, then a representation of each sub-image need be transmitted, can be handled with 8 bytes for each sub-problem. The original image file would be located on a network file server. Two byte integers for each of the following are required: the horizontal and vertical positions of the upper left corner of the sub-problem in the original image, the problem size (given a square problem), and the contextual region size (a contextual region is by definition square). If a requirement is later added for non-square problems, one additional integer would be required. Given the same 16 sub-problems as above, this would result in a transmission of only 128 bytes (16 x 8).

In this work, the third method was utilized. This resulted in the savings of 262,106 transmitted bytes over the first method, and a savings of 65,536 bytes over the second method. The trade-off is that each slave must load a copy of the original image from secondary storage.

3.4 Image Border

As mentioned in 1.4.2, current literature suggestions for handling the border are not valid. It is the opinion of the author that a user of a processed image would expect that the AHE algorithm technique should perform equally well in all areas of the image. It is widely known that most measuring devices are more accurate in their central regions than either end of their scales. However, that should not preclude every attempt to provide as much accuracy as possible throughout. The obvious solution for handling the inaccuracies of the image edges is to collect more data. This, besides being infeasible given existing images and the limitations of the data

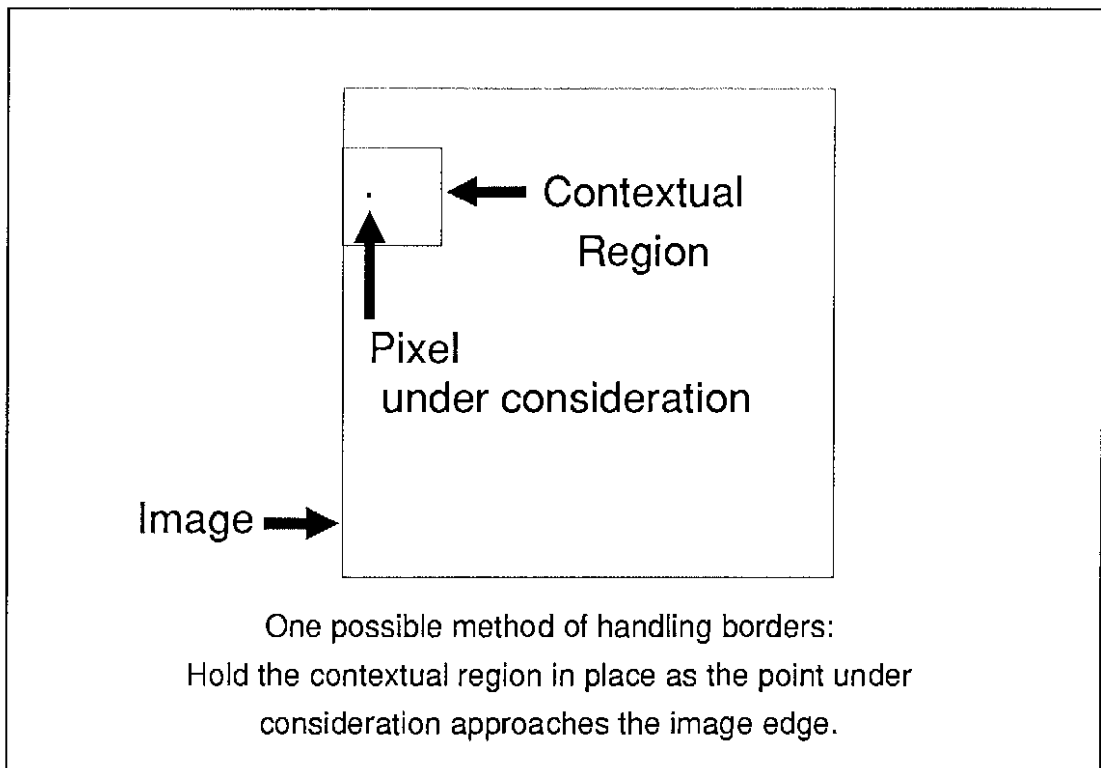


Figure 7: One possible method of handling borders.

collection devices, only delays the problem. There would then exist new edges with which to contend. The user would then expect accuracy near these edges as well.

One possible technique would be to hold the contextual region in place at the edge as the point under consideration approaches the edge (see Figure 7 above). This adds special cases to the algorithm. As the contextual region overlaps the edge, it must be 'backed up' to remain on the image. This must be considered in both the horizontal and vertical directions. There are special cases of left, right, top, bottom, upper-left, upper-right, lower-left, and lower-right. This introduces additional comparisons and referencing requirements into the algorithm implementation. Each additional computation requires time. Time is a precious commodity not freely available here.

Also, as the point of consideration moves off the center of the contextual region, values that are now more than half a width of a region away are being utilized in its computation. This also yields a bias.

Another technique would be to reduce the size of the contextual region. However as the size of the contextual region diminishes, the number of possible outcomes decreases. This results in an over-enhanced image near the edges. This would not be compatible with the remainder of the image. An example of varying the contextual region size is depicted in Figures 8 and 9 on page 19. This method also results in excessive comparisons and computations due to the necessity of detecting and handling special cases, thus resulting in a reduction of time performance.

The author has developed a method for mirroring the image across the border. Several benefits are derived from this technique. First, it eliminates special cases during the execution of the algorithm. An oversized image structure is built in primary storage with the original image placed in the center. The remainder of the structure is 'filled in' by mirroring the values across the image boundary. (See Figure 10 on page 20.) This eliminates numerous calculations. Second, each new pixel's value is computed by considering only values in its contextual region. That is, no compared pixel is more than half a region away in either the horizontal or vertical direction. Third, this eliminates the bias that occurs with a wrap-around technique.

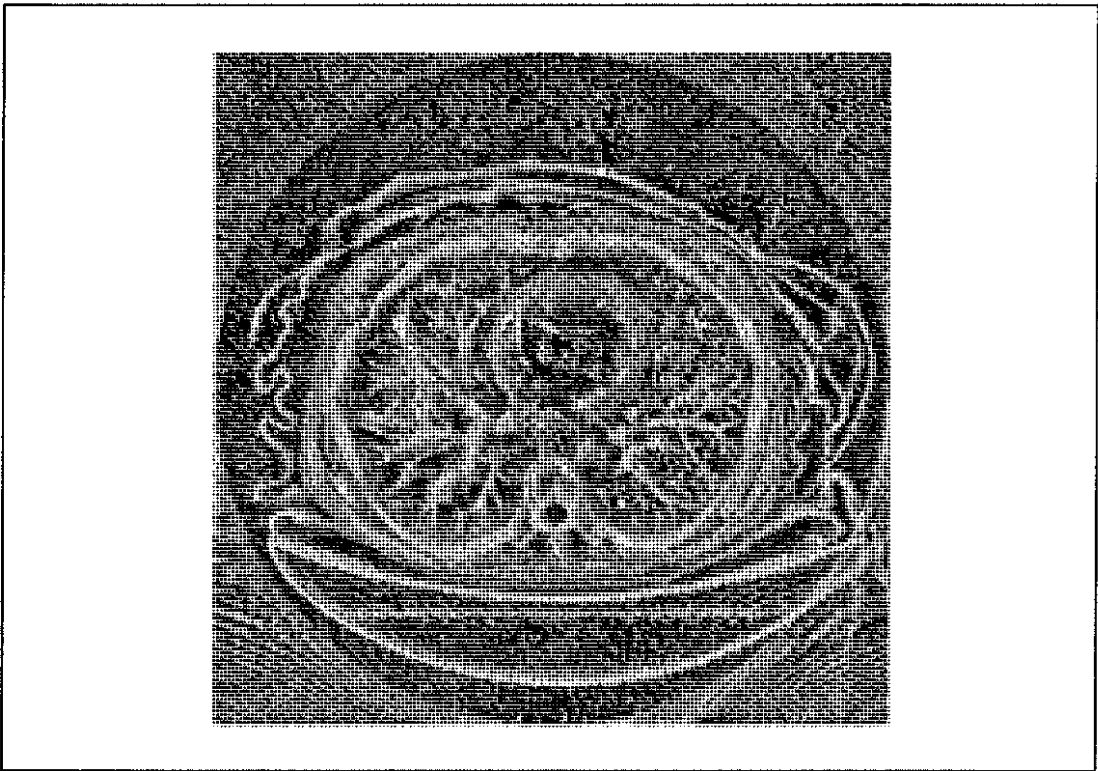


Figure 8: Chest - after AHE with contextual region of 16 x 16.

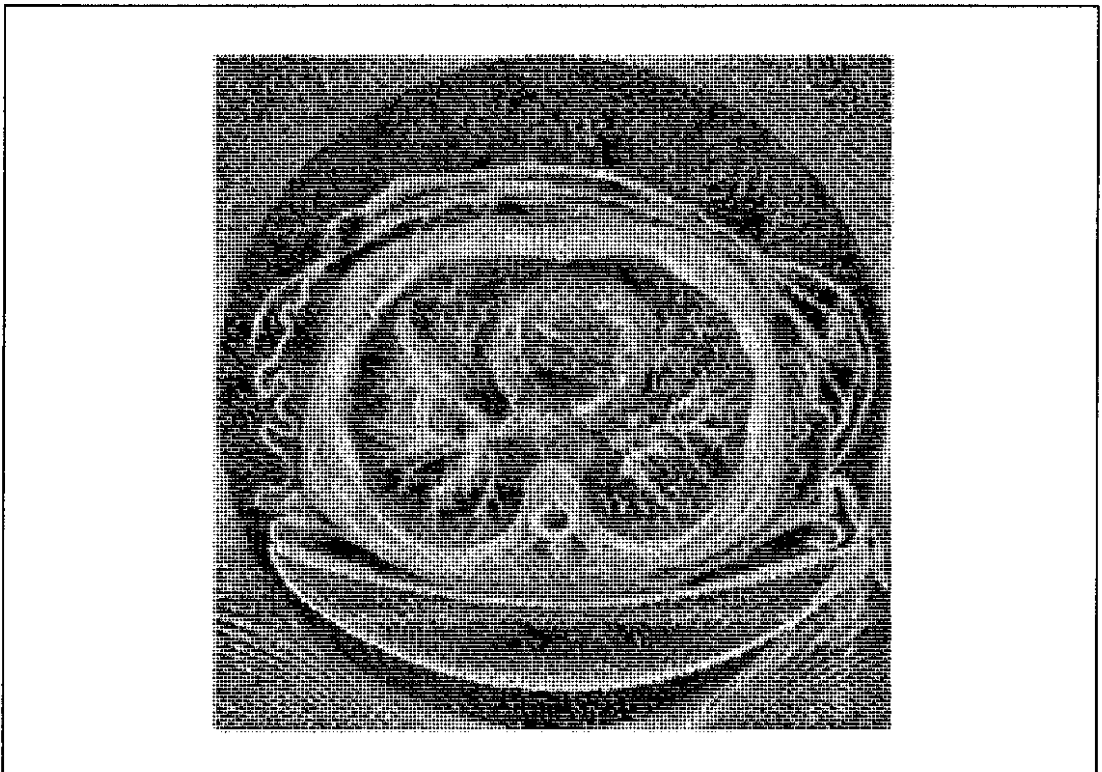


Figure 9: Chest - after AHE with contextual region of 32 x 32.

Original Image is located in center square.
 Array in primary storage includes mirrored border values.

1	27	26	25	25	26	27	28	29	30	31	32	32	31	30	8
20	1	18	17	17	18	19	20	21	22	23	24	24	23	8	21
12	11	1	9	9	10	11	12	13	14	15	16	16	8	14	13
4	3	2	1	1	2	3	4	5	6	7	8	8	7	6	5
4	3	2	1	1	2	3	4	5	6	7	8	8	7	6	5
12	11	10	9	9	10	11	12	13	14	15	16	16	15	14	13
20	19	18	17	17	18	19	20	21	22	23	24	24	23	22	21
28	27	26	25	25	26	27	28	29	30	31	32	32	31	30	29
36	35	34	33	33	34	35	36	37	38	39	40	40	39	38	37
44	43	42	41	41	42	43	44	45	46	47	48	48	47	46	45
52	51	50	49	49	50	51	52	53	54	55	56	56	55	54	53
60	59	58	57	57	58	59	60	61	62	63	64	64	63	62	61
60	59	58	57	57	58	59	60	61	62	63	64	64	63	62	61
52	51	57	49	49	50	51	52	53	54	55	56	56	54	54	53
44	57	42	41	41	42	43	44	45	46	47	48	48	47	64	45
57	35	34	33	33	34	35	36	37	38	39	40	40	39	38	64

Figure 10: Mirroring across the image boundaries.

3.5 Value Look-up Table (LUT)

As mentioned in Section 1.4.3, an issue that surfaced during the implementation of this work is the excessive number of computations in determining the new intensity values. Given an $m \times m$ contextual region, there are only m^2 possibilities for each new pixel intensity. Given an $n \times n$ image, this requires n^2 calculations to determine all of the new values. Since n^2 is much greater than m^2 , many of the calculations are repetitive. A Look-up Table has been employed to avoid the

redundant calculations. Since the new intensity values are required by each slave process, each slave builds its own LUT.

Another possibility was considered that would eliminate the need for a Look-up Table and the excessive multiplications and divisions. However, there needs to be some restrictions on the problem parameters. If we require the size of the contextual region to be a power of 2, and the number of intensities available on the display device also to be a power of 2, we can then reduce the calculations to register shifting. This may in fact be more expedient than a table look-up, however, for the purpose of the thesis and for general applicability, this method was not pursued.

This implementation under discussion takes a more general approach and does not limit the parameters to a specific set of values. The rationale behind this decision is that the size of the contextual region should be left to the user. Since the user needs to vary the region size to produce the best results as necessary, it would be overly restrictive to limit the available region sizes.

Whereas this discussion of excessive computations is not a feature of the parallel implementation, it is sound computing practice to streamline the sequential section of the code where possible. Thus it is mentioned here.

3.6 Coordination between Master and Slaves

The Message Passing Paradigm was outlined in Section 1.5.2. The following description depicts the actions of the master and slaves as applied to the AHE algorithm. A graphic representation is included in Figure 11 on page 23. The accompanying pseudo-code is included in Figures 12 and 13 on page 25.

The master is the first process to be invoked. It determines the number of slaves to be utilized by reading a file in secondary storage. This is done in accordance with the parallel processing program package [BUTLER90]. The size of the contextual region, the size of the problem pieces, and the names of the input and output files are read from the command line.

As the master invokes each slave, it establishes the lines of communication between itself and each slave. The master also sends the name of the input file to each slave process. The slaves perform their start-up work, i.e. loading the image and building the mirrored borders. The master by this time has divided the original problem into sub-problems as determined by the input parameters. A representation of each problem is placed in a linked list structure, referred to as the problem queue. The master then enters its main control loop. Each slave, upon completion of its initialization tasks, transmits a request for work to the master.

During the main control loop, the master checks to see if the condition exists that the slave queue is non-empty and the problem queue is non-empty. If these conditions are met, the master then dispatches a sub-problem to a slave represented

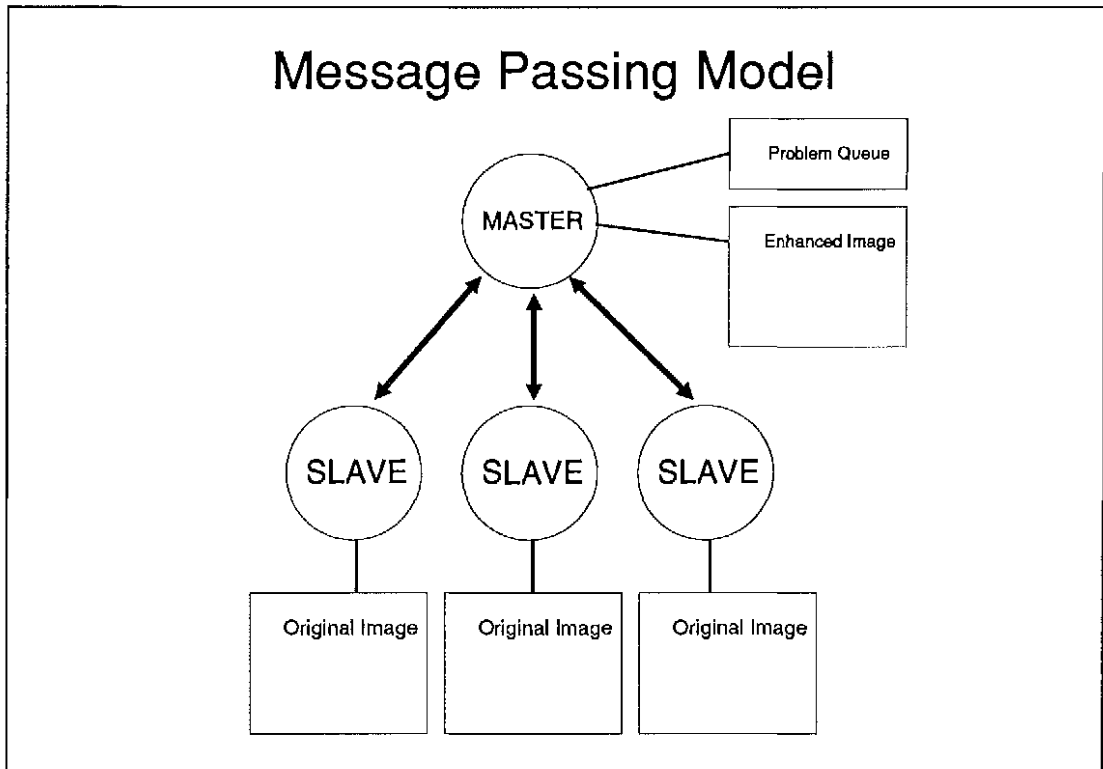


Figure 11: Message Passing Model.

on the slave queue. The appropriate piece of work and the slave are removed from their respective queues. As the master receives each request for work, it places a representation of the calling slave on a linked list slave queue.

The slave then receives the work, processes the sub-problem, and returns the resulting sub-image to the master. The master, upon receiving a message, determines the type of message, stores a copy of the sub-image in primary storage, then awaits the next message. This pattern continues until the master determines that all of the slaves are represented in the slave queue and the problem queue is empty. The problem has been completed at this point.

The master completes its functions by sending the completed image to secondary storage, issuing a command for the slaves to terminate, waiting for the slaves to terminate, and outputting any statistics gathered for testing purposes.

Master Process Pseudo-code

```
Create slaves
Read in parameters
Transmit control messages to slaves including image filename
Initialize solution data structure
Initialize slave queue
Initialize problem queue
Divide job into problems and place in problem queue
done = FALSE
While (NOT done)
{
  While (slave queue is NOT empty) and (problem queue is NOT empty)
    Send a problem to a slave
  Receive any message
  If message type is REQUEST_WORK then
    Add slave to slave queue
  If message type is SOLUTION then
    Update solution data structure
  If (ALL slaves are in slave queue) AND (problem queue is empty) then
    done = TRUE
}
Output solution to secondary storage
Send ENDSIGNAL message to slaves to terminate
Wait for all slaves to terminate
```

Figure 12: Master Process - Pseudo-Code.

Slave Process Pseudo-code

```
Receive message including image filename
Read image from secondary storage
Build mirrored borders
Send REQUEST_WORK message to master
while (received message type is NOT ENDSIGNAL)
{
  Receive a message
  If message type is DATA then
    Perform AHE on sub-problem described in message
  else
    Send ENDSIGNAL to next slave in loop
}
```

Figure 13: Slave Process - Pseudo-Code.

Chapter 4

TOOLS

Numerous software tools were utilized during the course of this work. Each of these were required to support the overall project. Their uses included presentation production, data conversion, and image viewing. These were in addition to the tools one would normally expect such as compilers, linkers, editors, etc. Several of these tools were developed by the author. These software endeavors were in addition to the main project implementation. Their description is included here to illustrate some of the additional research and development required by the author to complete this endeavor.

4.1 DISPLAY for the PC

One of the early research tools developed by the author was an interactive image processing program, DISPLAY. This was developed under MS-DOS for an IBM-PC clone with EGA graphics capabilities. Its original development was in Pascal. Later the software was converted to C. DISPLAY will load a 256 x 256 x 8 image from secondary storage into main memory. The image is then displayed in 16 color pseudo-color along with a histogram of the image.

DISPLAY is a menu-driven system that allows the user to perform point and area functions on the image. It will also allow several different output formats. The point operations include contrast enhancement, histogram stretching, and color

inversion. The area operations include neighborhood averaging (8 or 9 elements), neighborhood median (3 x 3 or 5 x 5), edge detection, Adaptive Histogram Equalization with choice of contextual region size, and several other filters. The output functions include formatting the currently displayed image as a raw image file (256 x 256 x 8), or in Postscript format, with or without accompanying histogram.

The DISPLAY software has been extremely beneficial in producing both hardcopy images as well as overhead slides for several presentations during the course of this work. The majority of the images included in this presentation were produced by DISPLAY.

4.2 'disp' for the Sun workstation with X Windows

During the project it was necessary to view the images being processed without the expense and time required to produce a hard copy. The author developed 'disp', a display program built in 'C' on the Sun workstation utilizing X Windows. It provided quick feedback during the development of the parallel AHE software. It was also beneficial to view the output images as the input parameters were varied.

4.3 Image and Histogram Production for Postscript

Some of the functions included in DISPLAY (See Section 4.1) were ported to the Unix environment. These included functions to produce Postscript files of images for presentation usage. These Postscript files were then 'viewed' either with

Ghostscript 1.3, the NeXT's Preview facility, or sent to a Postscript capable printer for hardcopy production. As Postscript is a language unto itself, it was necessary for the author to learn a subset of the language to facilitate Postscript file generation.

4.4 'usr/image' from UNC-CH

The medical images included as test data for this work were supplied by the UNC Medical Image Display Research Group. This research organization is located at the University of North Carolina at Chapel Hill. The format of some of the images is unique to UNC-CH. The data was converted to raw image files with the utilization of a suite of software tools known to UNC-CH as 'usr/image' running under X Windows. This software proved quite worthy in converting images between formats so as to enable the transfer of images between the author and members of the UNC-CH community.

4.5 Thesis publication software

In addition to the image production software mentioned above, the balance of this publication was produced with Harvard Graphics, Quattro-Pro, and Wordperfect.

Chapter 5

TESTING AND RESULTS

Program testing was conducted in a controlled manner allowing the varying of only one independent variable at a time. The variables that were controlled included number of slaves, size of the contextual region, and size of a sub-problem. A sequential version of the algorithm was extracted from the parallel slave code for a benchmark.

The sequential and parallel versions were executed on the same machine to eliminate disparities due to machine capabilities and performance tuning.

The software includes statistical data collection in the code for determining the elapsed time for each execution of the program. In addition, the Unix 'time' command was utilized to gather the same statistics.

Numerous timings were obtained at off-peak operating times. Testing was normally conducted at night after determining that system usage was at a minimum. The first series of 10 timings were taken on an 8-node Sequent Symmetry at the University of North Florida. A second series of 3 timings were taken on a 26-node Sequent Symmetry at Argonne National Laboratory in Chicago, IL.

The testing results have been compiled and are included in Appendix C.

Chapter 6

DISCUSSION OF RESULTS

The graphic results from the AHE algorithm were quite impressive. Both the chest image (see Figures 14 and 15 on page 31) and the brain image (see Figures 16 and 17 on page 32) benefitted greatly from the processing. However, the purpose of this work is not to verify the results of the image processing, but to improve the time performance.

As mentioned in Chapter 5, three independent variables were tested. These were the number of slave processes, the size of the contextual region, and the size of the sub-problem. The brain image was used for all testing. However, it should be noted that the choice of the image does not affect the performance of the algorithm. The numerical results are included in Appendix C. A graphic representation of these results are shown in Figures 18 through 22. Note that the sequential benchmark is indicated by the upright bar on the left of the graph, while the parallel timings are indicated by the lines on the chart. Each graph represents a single contextual region size. A separate line is included for each sub-problem size. The number of slaves utilized is indicated on the X-axis. The time in seconds is indicated on the Y-axis.

The results from the first set of timings, where the contextual region was set at 8 x 8, proved to be less than inviting. The amount of improvement hardly seemed worth the effort (see Figure 18 on page 33). The best case, for this contextual

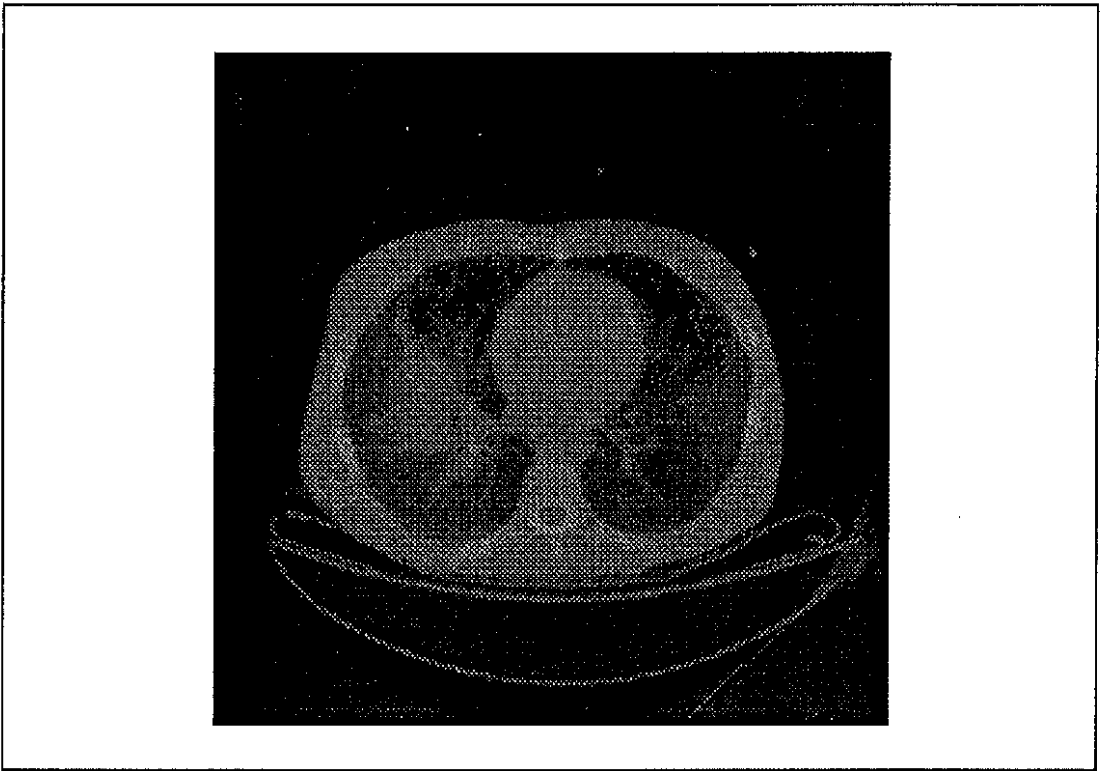


Figure 14: Chest - Original Image.

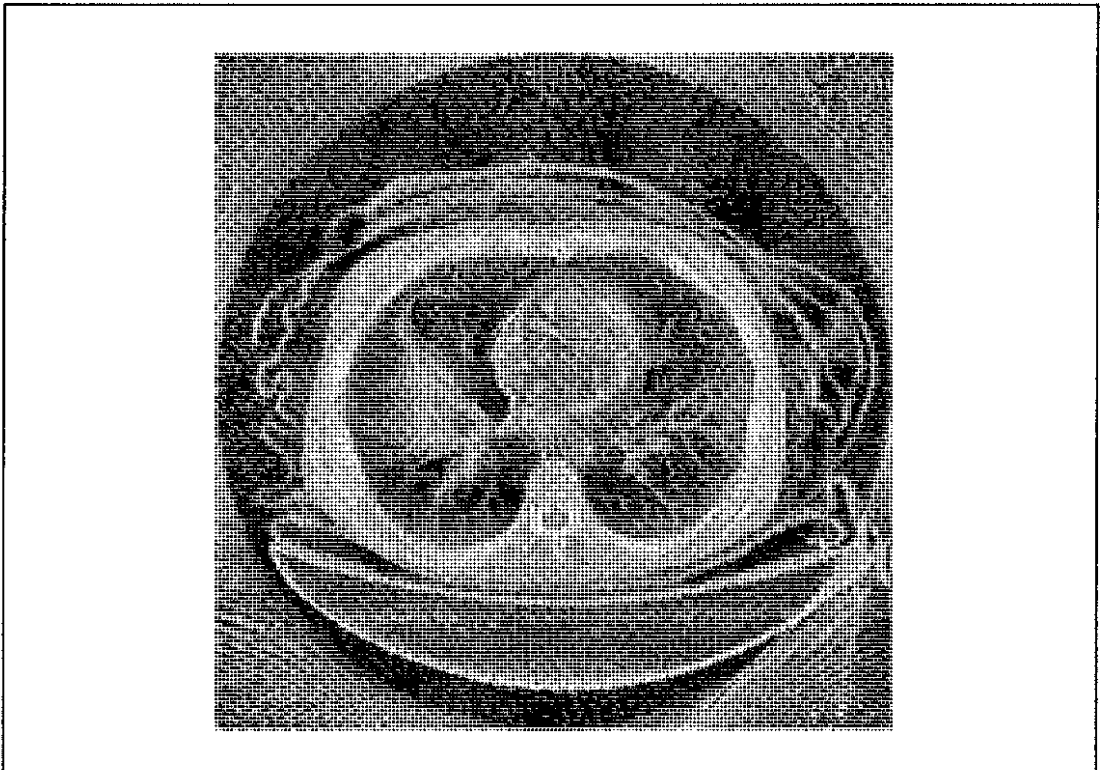


Figure 15: Chest - after AHE with contextual region of 64 x 64.

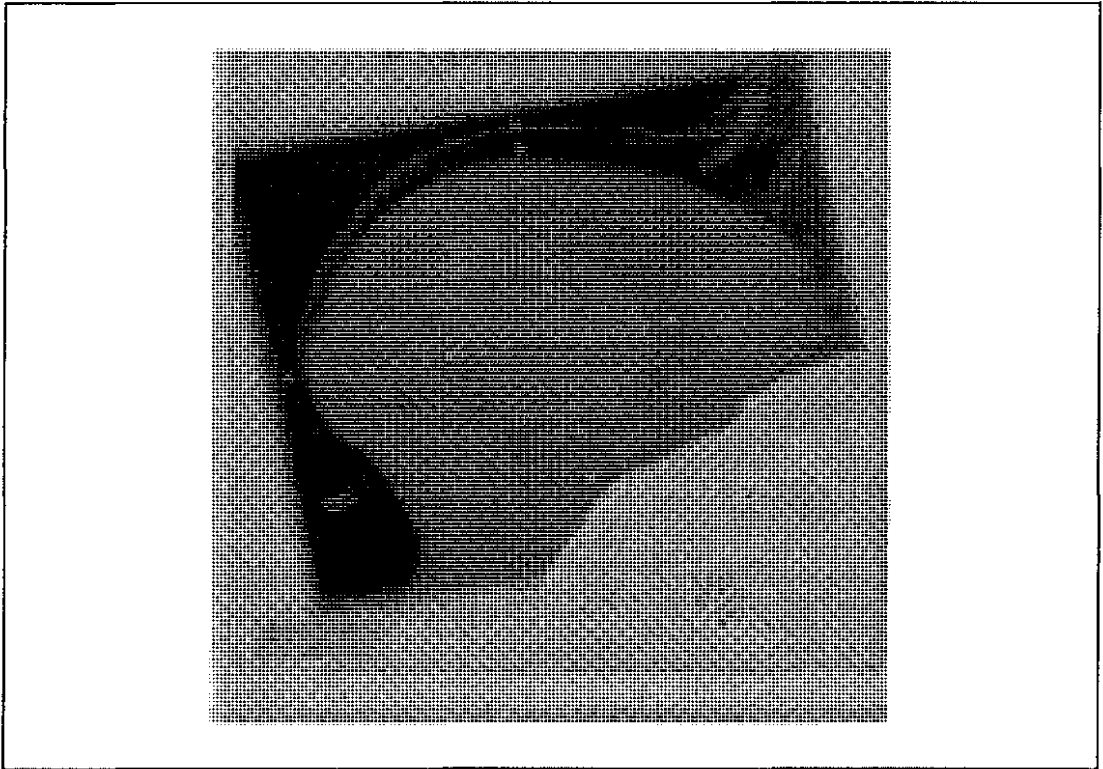


Figure 16: Brain - Original Image.

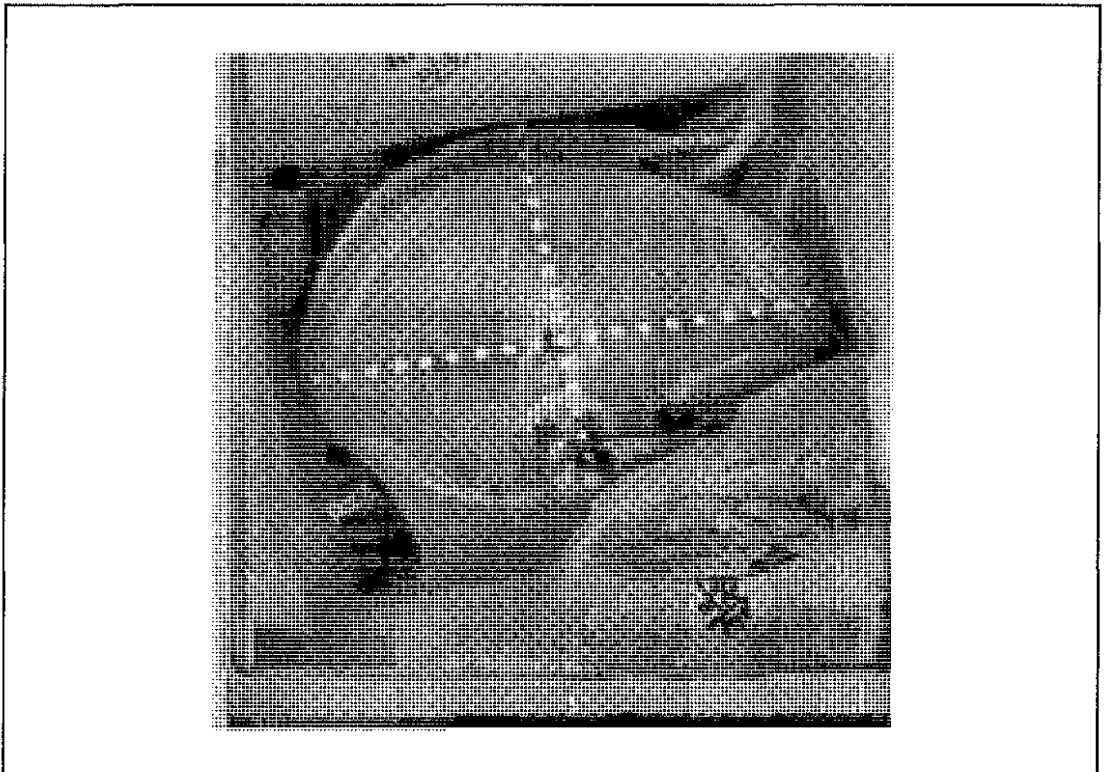


Figure 17: Brain - after AHE with contextual region of 64 x 64.

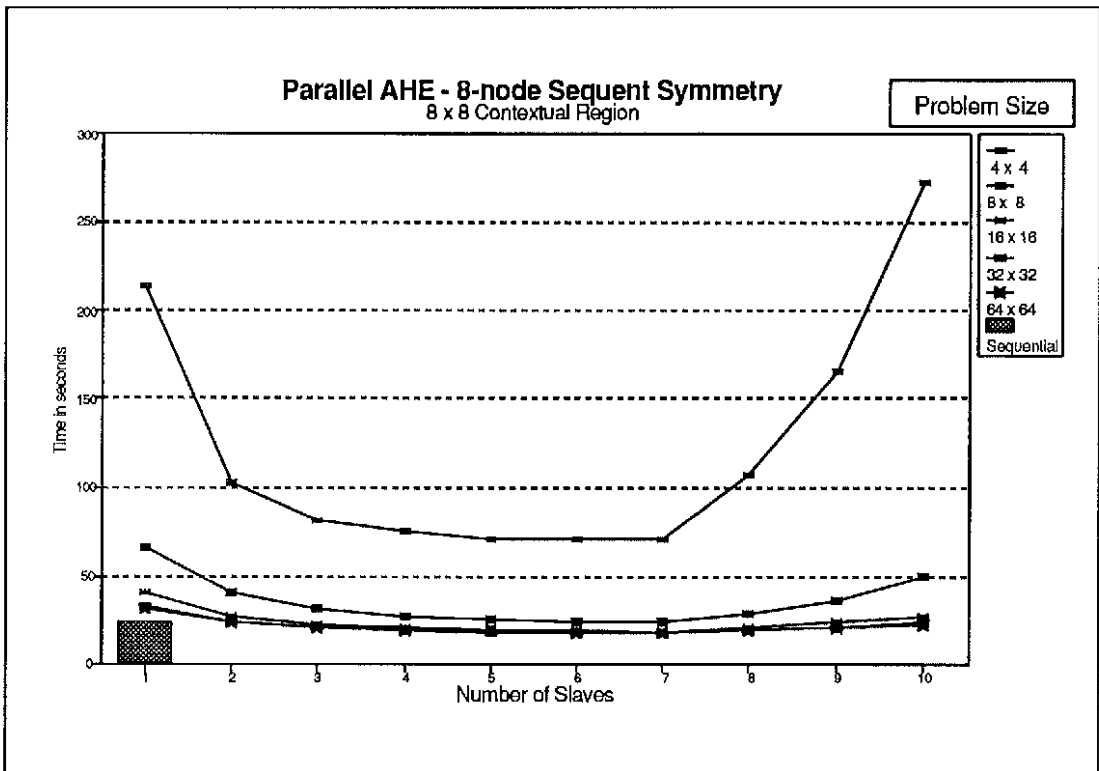


Figure 18: AHE Results - Contextual Region of 8 x 8.

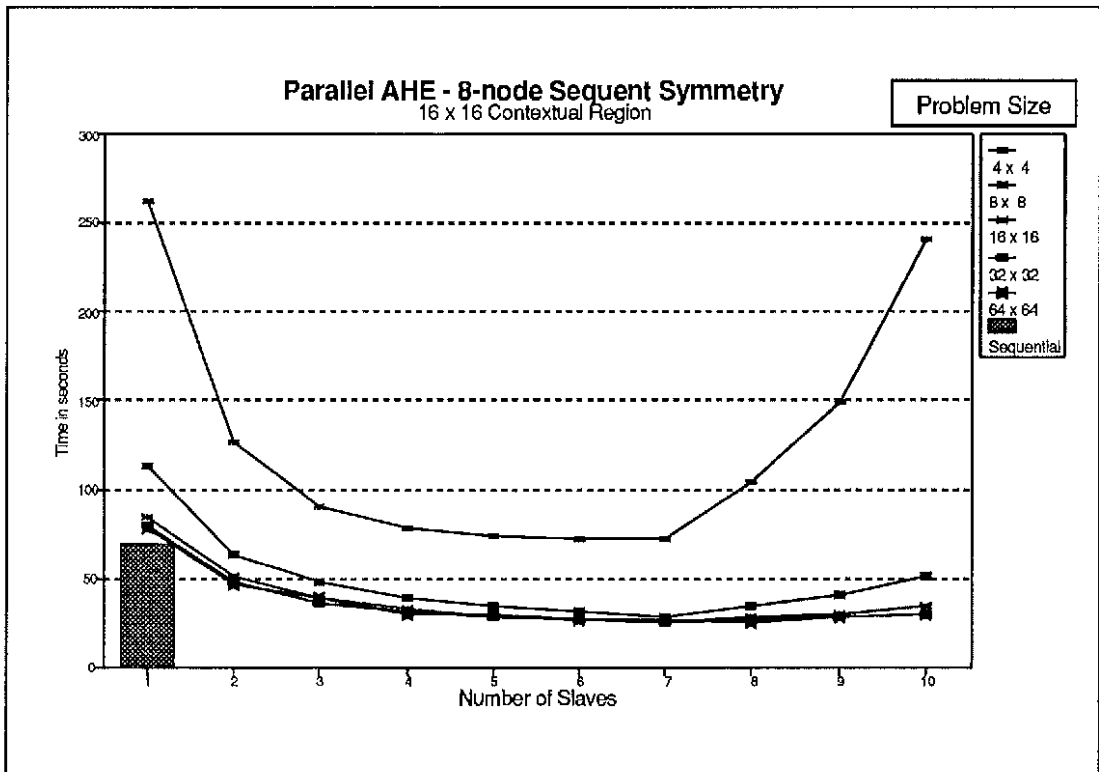


Figure 19: AHE Results - Contextual Region of 16 x 16.

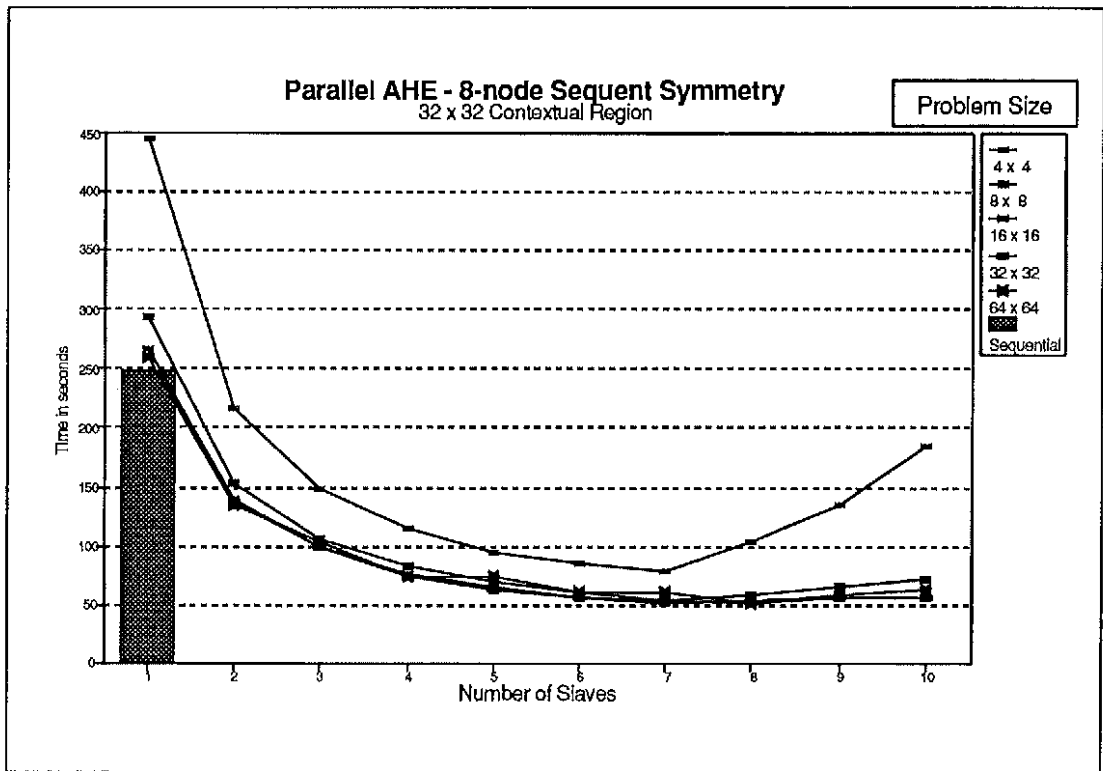


Figure 20: AHE Results - Contextual Region of 32 x 32.

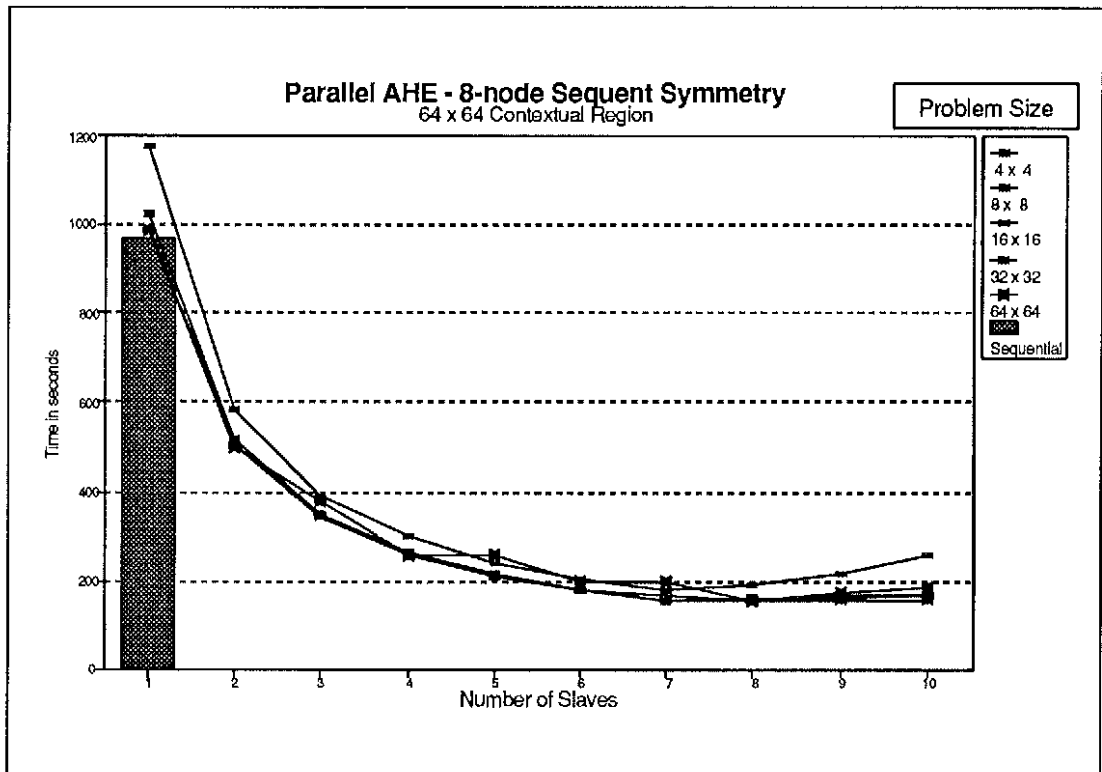


Figure 21: AHE Results - Contextual Region of 64 x 64.

region size, was where the sub-problem size was 64 x 64. The sequential benchmark was 23.8 seconds, while the parallel version with seven CPUs (a master and six slaves) completed the identical task in 17.3 seconds.

It is also noted that for extremely small sub-problems sizes, the performance was over an order of magnitude greater in some cases than for larger sub-problem sizes (see Figure 18 on page 33). Both of the above performances are due to the effect of granularity with respect to parallelism. Simply stated, if the problem is too small, any speed-up contributed by the parallelism will be more than offset by the coordination overhead. As the size of the contextual region is increased, the effect of parallelism slowly begins to show. With a contextual region of 32 x 32, a more desirable comparison with the sequential benchmark is obtained (see Figure 20 on page 34).

Increasing the contextual region size to 64 x 64 yields the best results with a speed-up of approximately 6.5:1 (see Figure 21 on page 34). According to Mr. Cromartie of UNC-CH, the largest popular contextual region size is one-sixteenth of the image size. Thus, the contextual region variable was limited to 64 x 64.

An interesting phenomena occurred as the number of slaves was increased. At first the results appear to approximate the function $1/n$ where n is the number of slaves. Whereas perfect speed-up can not be realized due to communication and coordination overhead, the curve is close. However, as the number of slaves surpassed 7 on the first set of timings, the execution time increased. Note that this

occurred at $n-1$ slaves where n is the number of available processors in the machine.

The maximum benefit is realized when each process has its own processor. As the number of slaves increases beyond $n-1$, there is more than one process sharing one or more of the processors. Multiple processes sharing a single processor increase overhead resulting in performance degradation.

The relationship between the number of processes and processors, however straightforward, seemed to beckon further support. With the assistance of Dr. Butler, the computing services at Argonne National Laboratory were obtained for additional testing. Due to the high amount of system usage at Argonne, the testing variables were limited. A contextual region of 64×64 was utilized, only varying the sub-problem size and the number of slaves. Also, testing for each possible slave count was bypassed in favor of sampling for the theorized trend. The resulting graph is included in Figure 22 on page 37.

It was estimated that the maximum performance would be at $n-1$ slaves where n was 26 for the 26-node Sequent Symmetry at Argonne. With the very small sub-problem size of 4×4 , the speed-up reversed at 18 slaves. Evidently the overhead was too great.

For the remaining cases, the trend reverses at about 22 to 24 slaves. This is lower than expected. However, it was noted that system usage was not minimal during any of the testing periods. Thus, some of the slaves spent time waiting for

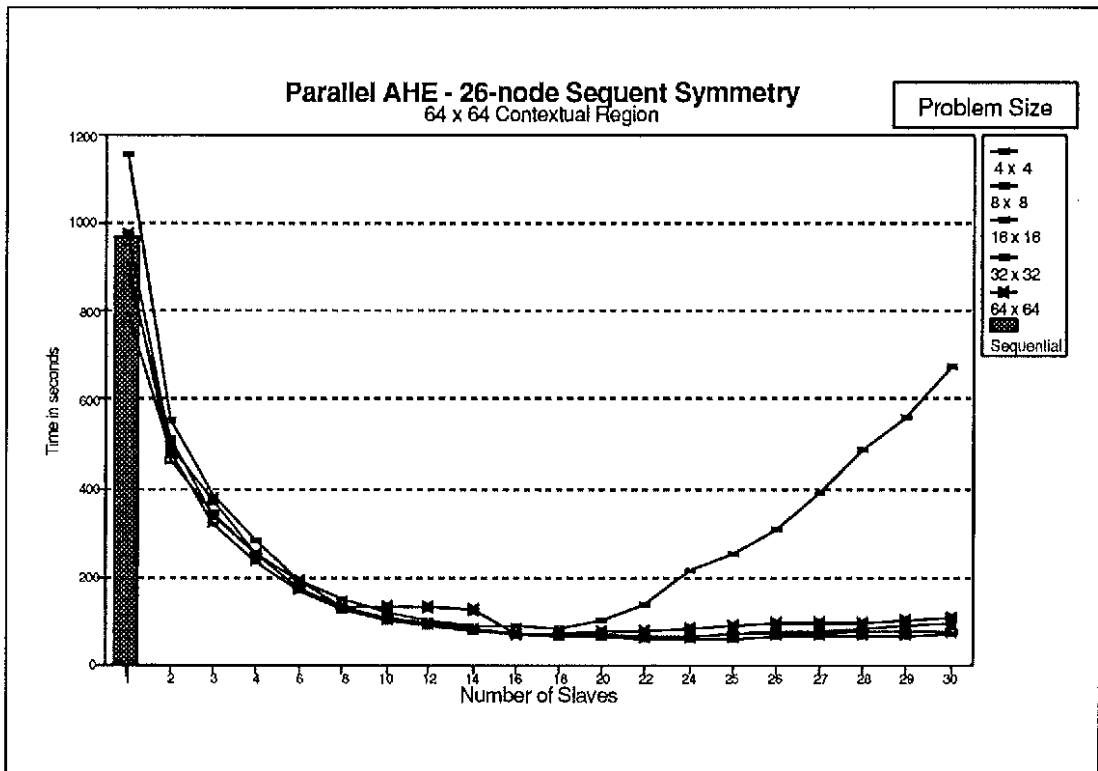


Figure 22: AHE Results - 26-node machine.

unrelated processes to relinquish processor control. This resulted in an overall increase in execution time.

Note however, that even though all 26 nodes were not available solely for the execution of this program, a worthwhile speedup was still realized. Where the 8-node machine yielded a best speed-up of approximately 6.5:1, the 26-node machine yielded a maximum speed-up of 16.5:1.

Chapter 7

CONCLUSIONS

The hypothesis of this work is restated here:

There are parallel processing techniques which will substantially improve the time performance of Adaptive Histogram Equalization.

The implementation and testing conducted supports the hypothesis.

In addition, a method for handling border regions was developed and implemented.

Also, the application of a Look-Up Table served well to reduce the number of required computations. Whereas this probably did not affect the outcome of the testing, as both the sequential and parallel versions utilized this method, it did serve as an improvement to the overall implementation.

Chapter 8

FUTURE WORK

There are numerous avenues that can be pursued to enhance this work. The research is by no means complete. One ultimate goal is to provide a real-time interactive system that would enable the user to manipulate the image as desired. Other goals are to enhance the algorithm to provide improved results. Several of these thoughts are included in the sections below. They are by no means a complete enumeration of the possibilities.

8.1 SIMD architecture

The parallel implementation of AHE in this thesis was performed on a Multiple Instruction Multiple Data (MIMD) machine. In this type of architecture, each processor can work on a problem independently or in coordination with other processors on the same machine. A different type of architecture is Single Instruction Multiple Data (SIMD). In this scenario each processor is executing the identical instruction in lock step on different data.

The SIMD machine requires a different design process during algorithm development and implementation. Different constraints are placed on the software. Current SIMD machines typically have less local memory per processing element than an MIMD machine. Accessing non-local memory is expensive. However, in

spite of some of these limitations, there may still be major advantages for utilizing this type of hardware.

8.2 Interactive viewing of image during processing

Due to the length of time required to process an image, it may be desirable to display partial results as the job is in progress. The user could invoke a process that would first display the original image. The user could then specify a contextual region size for the AHE processing. As the image is processed, the partial solutions could be super-imposed over the original image. The user would have the option of interrupting the process by specifying a new contextual region size, or terminating the process, or allowing the process to complete normally.

The method of problem subdivision included in this implementation is to utilize a checkerboard type pattern. The original image is simply divided into small squares and handed out for processing. Clearly this would not be an acceptable visual display as portions of the image were completed. A different method that would be easy to implement would be to have each problem contain every n th pixel. Then, as sub-problems are solved, the effect would be a more gradual and smooth transition of the image. This method is depicted in Figure 23 on page 41. Here the original image is divided into 16 sub-problems. Each pixel is labeled by its sub-problem number.

Dividing the image into interlaced sub-problems:

Each slave processes the set of pixels identified with a particular process number. As the results are returned and displayed, the effect is a gradual change vice a checkerboard effect depicted by the boxes.

1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
5	6	7	8	5	6	7	8	5	6	7	8	5	6	7	8
9	10	11	12	9	10	11	12	9	10	11	12	9	10	11	12
13	14	15	16	13	14	15	16	13	14	15	16	13	14	15	16
1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
5	6	7	8	5	6	7	8	5	6	7	8	5	6	7	8
9	10	11	12	9	10	11	12	9	10	11	12	9	10	11	12
13	14	15	16	13	14	15	16	13	14	15	16	13	14	15	16
1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
5	6	7	8	5	6	7	8	5	6	7	8	5	6	7	8
9	10	11	12	9	10	11	12	9	10	11	12	9	10	11	12
13	14	15	16	13	14	15	16	13	14	15	16	13	14	15	16
1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
5	6	7	8	5	6	7	8	5	6	7	8	5	6	7	8
9	10	11	12	9	10	11	12	9	10	11	12	9	10	11	12
13	14	15	16	13	14	15	16	13	14	15	16	13	14	15	16

Figure 23: Dividing the image into interlaced sub-problems.

8.3 The AHE Family of Algorithms

Research at the University of North Carolina at Chapel Hill has produced other members of the AHE family of algorithms [Pizer87][Zimmerman89]. Among them are Weighted Adaptive Histogram Equalization (WAHE) and Clipped Adaptive Histogram Equalization (CLAHE). Current work by Mr. Cromartie is being conducted on Adaptive Histogram Equalization using Edge Detection. Certainly, each of these may benefit from the application of parallelism. This is a field that needs further exploration.

8.4 Three-dimensional AHE

The advent of three-dimensional non-invasive imaging suggests the application of AHE in three dimensions. The concept of areas of low contrast in two dimensions can be expanded into the third dimension. As with two-dimensional AHE, this would be a pre-processing step prior to image display. The author has witnessed three-dimensional visualization tool development at UNC-CH and believes that 3-D AHE may be of benefit.

REFERENCES

[Austin87]

J.D. Austin, Stephen M. Pizer, "A Multiprocessor Adaptive Histogram Equalization Machine," The University of North Carolina at Chapel Hill, (1987).

[Boyle87]

J. Boyle, R. Butler, T. Disz, B. Glickfield, E. Lusk, R. Overbeek, J. Patterson, R. Stevens, Portable Programs for Parallel Processors, Holt, Rinehart and Winston, Inc., (1987).

[Butler90]

R. Butler, Unpublished technical memo, (1990).

[Ericksen90]

J.P. Ericksen, S.M. Pizer, J.D. Austin, "MAHEM: A Multiprocessor Engine for Fast Contrast-Limited Adaptive Histogram Equalization," Medical Imaging IV, SPIE Proceedings, 1233, (1990).

[Hummel75]

R.A. Hummel, "Histogram Modification Techniques," Computer Graphics and Image Processing, 4, (1975), pp. 209-224.

[Hummel77]

R.A. Hummel, "Image Enhancement by Histogram Transformation," Computer Graphics and Image Processing, 6, (1977), pp. 184-195.

[Ketcham76]

D.J. Ketcham, "Real-Time Image Enhancement Techniques," Seminar on Image Processing, Pacific Grove California, Hughes Aircraft Company, 1976, p. 1-6.

[Pizer81a]

S.M. Pizer, "An Automatic Intensity Mapping for the Display of CT Scans and Other Images", Medical Image Processing: Proceedings of the VIIth International Meeting on Information Processing in Medical Imaging, Stanford, California, Stanford University, (1981), pp. 276-309.

[Pizer81b]

S.M. Pizer, "Intensity Mappings for the Display of Medical Images", Functional Mapping of Organ Systems: 11th Annual Symposium on the Sharing of Computer Programs and Technology in Nuclear Medicine, New Orleans, Louisiana, P.D. Esser (ed.), Society of Nuclear Medicine, (1981), pp. 205-218.

[Pizer86]

S.M. Pizer, "Adaptive Histogram Equalization for Automatic Contrast Enhancement of Medical Images", Application of Optical Instrumentation in Medicine, XIV: Medical Imaging, Processing, and Display and Picture Archiving, and Communication Systems (PACS IV) for Medical Applications, (1986), pp. 242-250.

[Pizer87]

S.M. Pizer, "Adaptive Histogram Equalization and Its Variations", Computer Vision, Graphics, and Image Processing, 39, (1987), pp. 355-368.

[Zimmerman89]

J.B. Zimmerman, S.B. Cousins, M.E. Frisse, K.M. Hartzell, M.G. Kahn, "A Psychophysical Comparison of Two Methods for Adaptive Histogram Equalization," Washington University, St. Louis, Missouri 63130.

APPENDIX A

```
/* pmaster.c - source code */

/* Charles Kurak */
/* Master's THESIS - December, 1990 - University of North Florida */
/* Adaptive Histogram Equalization: A Parallel Implementation */

/* Thesis Committee:
/* Dr. Yap Chua - University of North Florida */
/* Dr. Ralph Butler - University of North Florida */
/* Dr. Charles Winton - University of North Florida */

#include <stdio.h>
#include "p4.h"
#include "p4_compat.h"

#define FALSE 0
#define TRUE 1
#define MAXVALUE 255
#define IMAGE_SIZE 256
#define SUBIMAGE_SIZE 64

/* message types */
#define CNTL 0
#define DATA 1
#define ENDSIGNAL 2

/* slave message types */
#define REQUEST_WORK 3
#define PROBLEM 4
#define SOLUTION 5

typedef struct procnode *PROCPTR;
struct procnode
{
    int procid;
    PROCPTR next;
};

struct data_msg_type {
    int type;
    int msg_type;
    int beg_x;
    int beg_y;
    int region_size;
    int x_size;
    int y_size;
    int max_value;
    unsigned char subimage[SUBIMAGE_SIZE][SUBIMAGE_SIZE];
};

typedef struct prob_node *PROBNODEPTR;

struct prob_node
{
    int beg_x;
    int beg_y;
};
```

```

    int region_size;
    int x_size;
    int y_size;
    int max_value;
    PROBNODEPTR next;
};

struct cntl_msg_type {
    int type;
    int prev;
    int next;
    char infilename[50];
};

struct endsignal_msg_type {
    int msg_type;
};

union d_e_msg {
    struct data_msg_type dmsg;
    struct endsignal_msg_type emsg;
};

int pbdone;
int nprocs;
int region_size;
int max_value = MAXVALUE;
int number_of_slaves;
PROBNODEPTR plist;
PROBNODEPTR freelist;
PROCPTR proclist;
PROCPTR freeproclist;

int gridsize;
int problem_size;
char infilename[50];
char outfilename[50];

unsigned char Master_Image[IMAGE_SIZE][IMAGE_SIZE];

/*****
** getprob()
*****/
getprob(prob)
union {
    struct data_msg_type dmsg;
    struct endsignal_msg_type emsg;
} *prob;
{
    int rc=1;
    int ctr;
    if ( (proclist_length() == number_of_slaves) && (plist == NULL))
    {
        rc=1; /* over by exhaustion */
    }
    else if (plist !=NULL)
    {
        /* make up problem message, taken from problem queue */
        (*prob).dmsg.type          = DATA;
        (*prob).dmsg.msg_type     = PROBLEM;
        (*prob).dmsg.beg_x        = plist->beg_x;
        (*prob).dmsg.beg_y        = plist->beg_y;
        (*prob).dmsg.region_size  = plist->region_size;
        (*prob).dmsg.x_size       = plist->x_size;
        (*prob).dmsg.y_size       = plist->y_size;
        (*prob).dmsg.max_value    = plist->max_value;
    }
}

```

```

    plist=plist->next;
    rc=0;
}
return (rc);
}

/*****
** make_node()
*****/
PROBNODEPTR make_node()
{
    PROBNODEPTR newptr;
    if((freelist) == NULL)
        newptr = (PROBNODEPTR) g_malloc(sizeof(struct prob_node));
    else
    {
        newptr = freelist;
        (freelist) = (freelist)->next;
    }
    if(newptr == NULL)
        printf("Out of Dynamic Memory!\n");
    else
    {
        int i;
        newptr->beg_x      = 0;
        newptr->beg_y      = 0;
        newptr->region_size = 0;
        newptr->x_size     = 0;
        newptr->y_size     = 0;
        newptr->max_value  = max_value;
        newptr->next       = NULL; /* set pointer to next node to NULL */
    }
    return newptr;
}

/*****
** probstart()
*****/
void probstart()
{
    PROBNODEPTR c;
    int i,j;

    for(j=0 ;j < IMAGE_SIZE ; j = j + problem_size)
    {
        for(i= 0 ; i < IMAGE_SIZE ; i = i + problem_size)
        {
            c = make_node();
            c->beg_x      = i;
            c->beg_y      = j;
            c->region_size = region_size;
            c->x_size     = problem_size;
            c->y_size     = problem_size;
            c->max_value  = max_value;
            c->next       = plist;
            plist        = c;
        }
    }
}

/*****
** make_proc_node()
*****/
PROCPTR make_proc_node()
{
    PROCPTR newptr;
    if((freeproclist) == NULL)

```

```

    newptr = (PROCPTR) g_malloc(sizeof(struct procnode));
else
{
    newptr = freeproclist;
    freeproclist = freeproclist->next;
}
if(newptr == NULL)
    printf("Out of Dynamic Memory\n");
else
{
    newptr->procid = 0; /* set value to zero */
    newptr->next = NULL; /* set pointer to next node to NULL */
}
return newptr;
}

/*****
** add_procnodeto_freeproclist()
*****/
void add_procnodeto_freeproclist(n)
PROCPTR n;
{
    n->procid=0;
    n->next = freeproclist;
    freeproclist = n;
}

/*****
** add_node_to_freelist()
*****/
void add_node_to_freelist(n)
PROBNODEPTR n;
{
    int ctr;
    n->beg_x      = 0;
    n->beg_y      = 0;
    n->region_size = 0;
    n->x_size     = 0;
    n->y_size     = 0;
    n->max_value  = 0;
    n->next      = freelist;
    freelist    = n;
}

/*****
** add_list_to_freelist()
*****/
void add_list_to_freelist(n)
PROBNODEPTR n;
{
    PROBNODEPTR cur;
    int ctr;
    if(n != NULL)
    {
        cur      = n;
        cur->beg_x      = 0;
        cur->beg_y      = 0;
        cur->region_size = 0;
        cur->x_size     = 0;
        cur->y_size     = 0;
        cur->max_value  = 0;
        while(cur->next != NULL)
            cur = cur->next;
        cur->next = freelist;
        freelist = n;
        n=NULL;
    }
}

```



```

    }
}

/*****
** dispose_freelist()
*****/
void dispose_freelist()
{
    PROBNODEPTR prev,cur;
    cur = freelist;
    prev=cur;
    while(cur != NULL)
    {
        prev = cur;
        cur = cur->next;
        g_free(prev);
    }
    freelist = NULL;
}

/*****
** dispose_freeproclist()
*****/
void dispose_freeproclist()
{
    PROCPTR prev,cur;
    cur = freeproclist;
    prev=cur;
    while(cur != NULL)
    {
        prev = cur;
        cur = cur->next;
        g_free(prev);
    }
    freeproclist = NULL;
}

/*****
** main()
*****/
main(argc,argv)
int argc;
char *argv[];
{
    struct cnll_msg_type rec1;
    struct data_msg_type rec2;
    union d_e_msg echo_data;
    union d_e_msg data;
    int id, i, msg_type;
    long timestart, timeend;

    struct endsignal_msg_type endsignal_msg;
    int ln;
    int slave();

    timestart=clock();

    if(argc!=6)
    {
        printf("Invalid number of input parameters. Program aborting.\n");
        exit(1);
    }

    plist = NULL;
    initenv(argc,argv);

```

```

create_procgrouop();
number_of_slaves=get_nslaves());

/* read in the contextual region size */
region_size=atoi(argv[1]);

/* read in the problem_size */
problem_size=atoi(argv[2]);

/* read in the max_value */
max_value=atoi(argv[3]);

/* read in the input filename */
strcpy(infile,argv[4]);

/* read in the output filename */
strcpy(outfile,argv[5]);

/* send control messages */
for (i=1; i < number_of_slaves+1; i++)
{
    rec1.type = CNTL;
    rec1.prev = (i-1);
    rec1.next = (i + 1) % (number_of_slaves+1);
    strcpy(rec1.infilename,infilename);
    ln = sizeof(struct cntl_msg_type);
    g_sendr(i,&rec1,sizeof(rec1));
}
/* initialize problem queue */
probstart();

pbdone=FALSE;
while ( !pbdone )
{
    while( (proclist_length() != 0) && (problist_length() != 0))
    {
        id=get_next_slave();
        getprob(&rec2);
        g_sendr(id,&rec2,sizeof(rec2));
    }
    g_rcv_any(&id,&data,&ln);
    msg_type=data.dmsg.msg_type;
    if (msg_type == REQUEST_WORK)
        add_slave_to_proclist(id);

    if (msg_type == SOLUTION)
        post_soln(&data);

    /* All the slaves are waiting for work && there is no more work */
    if((proclist_length()==number_of_slaves) && (problist_length()==0))
        pbdone=TRUE;
}

output_soln(outfile);

while(proclist_length()<number_of_slaves)
{
    g_rcv_any(&id,&data,&ln);
    msg_type=data.dmsg.msg_type;
    if (msg_type == REQUEST_WORK)
        add_slave_to_proclist(id);
}

/* end of problems & program */
endsignal_msg.msg_type = END SIGNAL;
g_sendr(1,&endsignal_msg,sizeof(endsignal_msg));

```

```

g_recv_any(&id,&echo_data,&ln);

dispose_freelist();
dispose_freeproclist();
add_list_to_freelist(plist);
plist = NULL;
wait_for_end();

timeend=clock();

printf("%2d %2d %2d %6.2f \n ",
number_of_slaves,region_size,problem_size,((float)(timeend - timestart))/100);
}

/*****
** get_next_slave()
*****/
get_next_slave()
{
    PROCPTR cur;
    int rc=0;
    if(proclist != NULL)
    {
        cur=proclist;
        rc=cur->procid;
        proclist=proclist->next;
        add_procnodeto_freeproclist(cur);
    }
    return rc;
}

/*****
** add_slave_to_proclist()
*****/
add_slave_to_proclist(i)
int i;
{
    PROCPTR sq;
    sq=make_proc_nodc();
    sq->procid=i;
    sq->next=proclist;
    proclist=sq;
}

/*****
** proclist_length()
*****/
proclist_length()
{
    int i=0;
    PROCPTR cur;
    cur = proclist;
    while (cur)
    {
        i++;
        cur=cur->next;
    }
    return i;
}

/*****
** problast_length()
*****/
problast_length()
{
    int i=0;

```

```

PROBNODEPTR cur;
cur = plist;
while (cur)
{
    i++;
    cur=cur->next;
}
return i;
}

/*****
** post_prob()
*****/
post_prob(c)
union d_e_msg *c;
{
    PROBNODEPTR d;
    int ctr;
    d
        = make_node();
    d->beg_x
        = (*c).dmsg.beg_x;
    d->beg_y
        = (*c).dmsg.beg_y;
    d->region_size
        = (*c).dmsg.region_size;
    d->x_size
        = (*c).dmsg.x_size;
    d->y_size
        = (*c).dmsg.y_size;
    d->max_value
        = max_value;
    d->next
        = plist;
    plist
        = d;
}

/*****
** post_soln()
*****/
post_soln(c)
union d_e_msg (*c);
{
    /* copy values from message into Master_Image[]; **
    int x,y;
    int i,j;
    int beg_x;
    int beg_y;
    int x_size;
    int y_size;
    int max_x;
    int max_y;

    beg_x = (*c).dmsg.beg_x;
    beg_y = (*c).dmsg.beg_y;
    x_size = (*c).dmsg.x_size;
    y_size = (*c).dmsg.y_size;

    max_x = beg_x + x_size;
    max_y = beg_y + y_size;

    for(j=0,y=beg_y;y < max_y;y++,j++)
        for(x=beg_x,i=0;x < max_x;x++,i++)
            Master_Image[y][x] = (*c).dmsg.subimage[j][i];
}

/*****
** output_soln()
*****/
output_soln(outfilename)
char outfilename[];
{
    int i,j;
    FILE *data;

```

```
data = fopen(outfilename,"wb");

for(j=0;j<IMAGE_SIZE;j++)
  for(i=0;i<IMAGE_SIZE;i++)
    fprintf(data,"%c",Master_Image[j][i]);

fclose(data);
}

/*****
** EOF pmaster.c
*****/
```

APPENDIX B

```
/* pslave.c - source code */

/* Charles Kurak */
/* Master's THESIS - December, 1990 - University of North Florida */
/* Adaptive Histogram Equalization: A Parallel Implementation */

/* Thesis Committee:
/* Dr. Yap Chua - University of North Florida */
/* Dr. Ralph Butler - University of North Florida */
/* Dr. Charles Winton - University of North Florida */

#include "p4.h"
#include "p4_compat.h"

#include <stdio.h>

#define FALSE 0
#define TRUE 1

#define BORDER 128
#define IMAGE_SIZE 256
#define ARRAY_SIZE (IMAGE_SIZE + (BORDER * 2))

#define SOLUTION_SIZE 64

/* message types */
#define CNTL 0
#define DATA 1
#define ENDSIGNAL 2

/* slave message types */
#define REQUEST_WORK3
#define SOLUTION 5

struct cntl_msg_type {
    int type;
    int prev;
    int next;
    char infilename[ 50 ];
};

struct data_msg_type {
    int type;
    int msg_type;
    int beg_x;
    int beg_y;
    int region_size;
    int problem_size_x;
    int problem_size_y;
    int max_value;
    unsigned char subimage[ SOLUTION_SIZE ][ SOLUTION_SIZE ];
};

struct endsignal_msg_type {
    int msg_type;
};
```

```

union d_e_msg {
    struct data_msg_type dmsg;
    struct endsignal_msg_type emsg;
};

char infile[50];

unsigned char Array[ ARRAY_SIZE ][ ARRAY_SIZE ];
unsigned char LUT[ SOLUTION_SIZE * SOLUTION_SIZE ];

/*****
** slave()
*****/
slave()
{
    int master_id;
    int msg_type;
    struct cntl_msg_type rec1;
    union d_e_msg rec2;
    int ln;
    struct endsignal_msg_type endsignal_msg;

    /* Get successor's id */
    g_rcv_any( &master_id, &rec1, &ln );
    msg_type = rec1.type;
    strcpy( infile, rec1.infilename );

    build_image( infile );

    rec2.dmsg.msg_type = REQUEST_WORK;
    g_sndr( master_id, &rec2, sizeof( rec2 ) );

    while (msg_type != ENDSIGNAL)
    {
        g_rcv_any( &master_id, &rec2, &ln );
        msg_type = rec2.dmsg.type;

        if (msg_type == DATA)
        {
            work( &rec2 );

            /* send work request to master */
            rec2.dmsg.msg_type = REQUEST_WORK;
            g_sndr( master_id, &rec2, sizeof( rec2 ) );
        }
        else
        {
            endsignal_msg.msg_type = ENDSIGNAL;
            g_sndr( rec1.next, &endsignal_msg, sizeof( endsignal_msg ) );
        }
    }
}

/*****
** work()
*****/
work( rec )
union
{
    struct data_msg_type dmsg;
    struct endsignal_msg_type emsg;
} *rec;
{
    int Image_beg_x;
    int Image_beg_y;
    int Array_beg_x;

```

```

int Array_beg_y;
int region_size;
int problem_size_x;
int problem_size_y;
int Array_end_x;
int Array_end_y;
int max_value;
int half_region_size;

struct {
    int beg_x;
    int beg_y;
    int problem_size_x;
    int problem_size_y;
    unsigned char subimage[ SOLUTION_SIZE ][ SOLUTION_SIZE ];
} soln;

register int i,j,x,y;
int min_region_x;
int max_region_x;
int min_region_y;
int max_region_y;
long rank;
unsigned char localvalue;
long region_area;

Image_beg_x    = (*rec).dmsg.beg_x;
Image_beg_y    = (*rec).dmsg.beg_y;
region_size    = (*rec).dmsg.region_size;
problem_size_x = (*rec).dmsg.problem_size_x;
problem_size_y = (*rec).dmsg.problem_size_y;
max_value      = (*rec).dmsg.max_value;

Array_beg_x    = Image_beg_x + BORDER;
Array_beg_y    = Image_beg_y + BORDER;
Array_end_x    = Array_beg_x + problem_size_x;
Array_end_y    = Array_beg_y + problem_size_y;
region_area    = region_size * region_size;
half_region_size = (region_size / 2);

/* Build Look-up Table */
build_LUT( max_value, region_area );

for( y = Array_beg_y ; y < Array_end_y ; y++ )
{
    for( x = Array_beg_x ; x < Array_end_x ; x++ )
    {
        localvalue = Array[ y ][ x ];
        rank = 0;
        /* compute little histogram boundaries */
        min_region_x = x - half_region_size;
        max_region_x = x + half_region_size;
        min_region_y = y - half_region_size;
        max_region_y = y + half_region_size;

        /* Calculate a pixel's rank */
        for( j = min_region_y ; j < max_region_y ; j++ )
            for( i = min_region_x ; i < max_region_x ; i++ )
                if ( Array[ j ][ i ] < localvalue )
                    rank++;

        /* assign new value to pixel using Look-Up-Table */
        soln.subimage[y-Array_beg_y][x-Array_beg_x] = LUT[ rank ];
    }
}

```



```

/* send solution to master */
soln.beg_x      = (*rec).dmsg.beg_x;
soln.beg_y      = (*rec).dmsg.beg_y;
soln.problem_size_x = problem_size_x;
soln.problem_size_y = problem_size_y;
send_soln(&soln);
}

/*****
** send_soln()
*****/
send_soln(soln)
struct
{
    int beg_x;
    int beg_y;
    int problem_size_x;
    int problem_size_y;
    unsigned char subimage[ SOLUTION_SIZE ][ SOLUTION_SIZE ];
} *soln;
{
    union d_e_msg msg;
    int i;
    int j;
    int master_id=0;

    msg.dmsg.type      = DATA;
    msg.dmsg.msg_type  = SOLUTION;
    msg.dmsg.beg_x     = soln->beg_x;
    msg.dmsg.beg_y     = soln->beg_y;
    msg.dmsg.region_size = 0;
    msg.dmsg.problem_size_x = soln->problem_size_x;
    msg.dmsg.problem_size_y = soln->problem_size_y;
    msg.dmsg.max_value = 0;
    for( i = 0; i < SOLUTION_SIZE; i++ )
        for( j = 0; j < SOLUTION_SIZE; j++ )
            msg.dmsg.subimage[ i ][ j ] = (*soln).subimage[ i ][ j ];

    /* send partial or complete solution to master */
    g_sendr( master_id, &msg, sizeof( msg ) );
}

/*****
** build_image()
*****/
build_image(infile)
char *infile;
{
    FILE *data;
    int y=0;
    int x,i,j;

    /* Clear Array */
    for( y = 0; y < ARRAY_SIZE; y++ )
        for( x = 0; x < ARRAY_SIZE; x++ )
            Array[ y ][ x ] = 0;

    data = fopen(infile,"r");
    for( y = BORDER; y < BORDER + 256; y++ )
        fread( Array[ y ] + BORDER, 256, 1, data );
    fclose( data );

    /* Build Border */
    for( j = BORDER; j < IMAGE_SIZE + BORDER; j++ )

```

```

for( i = 0; i < BORDER; i++ )
{
    /* Mirror left side */
    Array[ j ][ BORDER - i - 1 ] = Array[ j ][ BORDER + i ];

    /* Mirror right side */
    Array[ j ][ BORDER + IMAGE_SIZE + i ] =
        Array[ j ][ BORDER + IMAGE_SIZE - i - 1 ];

    /* Mirror top side */
    Array[ BORDER - i - 1 ][ j ] = Array[ BORDER + i ][ j ];

    /* Mirror bottom side */
    Array[ BORDER + IMAGE_SIZE + i ][ j ] =
        Array[ BORDER + IMAGE_SIZE - i - 1 ][ j ];
}

/* Lower upper left */
for( y = 0; y < BORDER - 1; y++ )
    for( x = 0; x < BORDER - y - 1; x++ )
        Array[ BORDER - y - 1 ][ x ] = Array[ BORDER + y ][ x ];

/* Lower upper right */
for( y = 0; y < BORDER - 1; y++ )
    for( x = y + 1; x < BORDER; x++ )
        Array[ BORDER - y - 1 ][ BORDER + IMAGE_SIZE + x ] =
            Array[ BORDER + y ][ BORDER + IMAGE_SIZE + x ];

/* upper lower left */
for( y = 0; y < BORDER - 1; y++ )
    for( x = 0; x < BORDER - y - 1; x++ )
        Array[ IMAGE_SIZE + BORDER + y ][ x ] =
            Array[ IMAGE_SIZE + BORDER - y - 1 ][ x ];

/* upper lower right */
for( y = 0; y < BORDER - 1; y++ )
    for( x = y + 1; x < BORDER; x++ )
        Array[ IMAGE_SIZE + BORDER + y ][ BORDER + IMAGE_SIZE + x ] =
            Array[ IMAGE_SIZE + BORDER - y - 1 ][ BORDER + IMAGE_SIZE + x ];

/* Upper upper left */
for( y = 0; y < BORDER; y++ )
    for( x = 0; x < y + 1; x++ )
        Array[ BORDER - y - 1 ][ BORDER - x ] =
            Array[ BORDER + y ][ BORDER - x ];

/* Lower lower left */
for( y = 0; y < BORDER; y++ )
    for( x = 0; x < y + 1; x++ )
        Array[ IMAGE_SIZE + BORDER + y ][ BORDER - x ] =
            Array[ IMAGE_SIZE + BORDER - y - 1 ][ BORDER - x ];

/* Upper upper right */
for( y = 0; y < BORDER; y++ )
    for( x = 0; x < y; x++ )
        Array[ BORDER - y - 1 ][ IMAGE_SIZE + BORDER + x ] =
            Array[ BORDER + y ][ IMAGE_SIZE + BORDER + x ];

/* Lower lower right */
for( y = 0; y < BORDER; y++ )
    for( x = 0; x < y; x++ )
        Array[ IMAGE_SIZE + BORDER + y ][ IMAGE_SIZE + BORDER + x ] =
            Array[ IMAGE_SIZE + BORDER - y - 1 ][ IMAGE_SIZE + BORDER + x ];

/* Upper left Diagonal */

```

```

for( y = 0; y < BORDER; y++ )
    Array[ y ][ y ] = Array[ BORDER ][ BORDER ];

/* Lower right Diagonal */
for( y = 0; y < BORDER; y++ )
    Array[ IMAGE_SIZE + BORDER + y ][ BORDER + IMAGE_SIZE + y ] =
        Array[ IMAGE_SIZE + BORDER - 1 ][ IMAGE_SIZE + BORDER - 1 ];

/* Lower left Diagonal */
for( y = 0; y < BORDER; y++ )
    Array[ IMAGE_SIZE + BORDER + BORDER - y - 1 ][ y ] =
        Array[ IMAGE_SIZE + BORDER ][ BORDER ];

/* Upper right Diagonal */
for( y = 0; y < BORDER; y++ )
    Array[ BORDER - y - 1 ][ y + IMAGE_SIZE + BORDER ] =
        Array[ BORDER ][ IMAGE_SIZE + BORDER ];
}

/*****
** build_LUT()
*****/
build_LUT( max_value, region_area )
int max_value;
int region_area;
{
    int i;
    static int j = 0;
    static int k = 0;

    /* A different max_value or region_area requires a different LUT */
    /* Only build a new LUT if a change has been made requiring a new LUT */
    if( (j != max_value) || (k != region_area) )
    {
        j = max_value;
        k = region_area;
        for( i = 0; i < region_area; i++)
            LUT[ i ] = (unsigned char) (( i * max_value ) / ( region_area - 1 ));
    }
}

/*****
** EOF pslave.c
*****/

```

APPENDIX C

Test Results: Contextual Region of 8 x 8.

Hardware: UNF Sequent Symmetry
(8-node machine)

<u>Sequential</u>	Number of <u>Slaves</u>	Parallel Problem Size				
		<u>4x4</u>	<u>8x8</u>	<u>16x16</u>	<u>32x32</u>	<u>64x64</u>
23.8	1	214.5	66.2	39.5	33.0	31.7
	2	102.2	39.8	26.5	23.6	22.9
	3	80.5	31.2	22.5	20.6	20.2
	4	74.7	27.3	20.5	18.6	18.7
	5	70.9	24.8	18.9	18.1	18.6
	6	71.0	23.8	18.5	17.7	17.3
	7	70.9	23.4	17.9	17.4	17.5
	8	106.9	28.7	20.3	19.0	18.7
	9	165.7	35.4	23.4	21.0	20.2
	10	272.4	49.5	26.6	22.9	22.3

All times in seconds.

Table 1: 8-node Sequent Symmetry, 8 x 8 Contextual Region.

Test Results: Contextual Region of 16 x 16.

Hardware: UNF Sequent Symmetry
(8-node machine)

<u>Sequential</u>	Number of <u>Slaves</u>	Parallel Problem Size				
		<u>4x4</u>	<u>8x8</u>	<u>16x16</u>	<u>32x32</u>	<u>64x64</u>
69.9	1	262.3	112.2	84.9	79.0	77.5
	2	126.7	62.3	50.6	47.4	46.9
	3	89.8	47.3	39.1	36.3	38.1
	4	77.8	38.5	32.1	30.6	30.2
	5	73.4	33.8	28.8	27.9	30.3
	6	71.4	30.7	26.5	26.0	26.4
	7	71.4	28.6	24.9	24.4	26.6
	8	103.3	33.7	27.4	26.1	25.0
	9	149.3	39.5	30.1	27.8	27.6
	10	240.9	50.7	34.2	29.3	30.3

All times in seconds.

Table 2: 8-node Sequent Symmetry, 16 x 16 Contextual Region.

Test Results: Contextual Region of 32 x 32.

Hardware: UNF Sequent Symmetry
(8-node machine)

<u>Sequential</u>	Number of <u>Slaves</u>	Parallel Problem Size				
		<u>4x4</u>	<u>8x8</u>	<u>16x16</u>	<u>32x32</u>	<u>64x64</u>
251.0	1	445.0	293.9	266.8	260.5	259.4
	2	217.2	153.6	139.6	136.8	135.5
	3	149.2	106.4	97.9	97.9	104.5
	4	115.1	83.0	76.4	74.8	74.4
	5	95.5	69.1	64.2	63.3	74.6
	6	85.2	59.8	55.7	55.6	60.6
	7	79.6	54.4	51.2	52.7	60.0
	8	103.0	58.2	52.9	52.3	51.6
	9	135.7	64.0	54.8	55.9	57.1
	10	184.5	71.0	56.8	56.7	62.3

All times in seconds.

Table 3: 8-node Sequent Symmetry, 32 x 32 Contextual Region.

Test Results: Contextual Region of 64 x 64.

Hardware: UNF Sequent Symmetry
(8-node machine)

<u>Sequential</u>	Number of <u>Slaves</u>	Parallel Problem Size				
		<u>4x4</u>	<u>8x8</u>	<u>16x16</u>	<u>32x32</u>	<u>64x64</u>
974.5	1	1174.3	1024.7	997.2	989.1	987.5
	2	583.7	518.1	505.3	502.3	500.4
	3	394.1	349.3	341.5	347.3	376.6
	4	298.4	265.5	259.0	257.2	256.6
	5	241.7	214.7	211.9	210.5	255.0
	6	203.4	180.9	178.3	180.1	194.9
	7	177.0	157.4	154.8	164.8	195.3
	8	193.1	159.6	155.4	155.8	154.9
	9	217.7	162.1	156.7	166.1	173.0
	10	256.6	166.0	157.4	165.2	187.8

All times in seconds.

Table 4: 8-node Sequent Symmetry, 64 x 64 Contextual Region.

Test Results: Contextual Region: 64 x 64.

Hardware: Argonne National Laboratory Sequent Symmetry
(26-node machine)

<u>Sequential</u>	Number of <u>Slaves</u>	Parallel Problem Size				
		<u>4x4</u>	<u>8x8</u>	<u>16x16</u>	<u>32x32</u>	<u>64x64</u>
974.5	1	1159.8	882.7	794.3	915.4	979.2
	2	556.0	512.5	481.4	462.2	494.8
	3	386.5	335.7	318.1	341.2	371.1
	4	280.6	253.9	233.6	250.5	250.6
	6	190.1	172.8	165.0	173.9	189.4
	8	147.5	126.9	126.0	128.9	128.6
	10	118.4	101.2	102.9	105.1	128.3
	12	99.7	88.1	86.3	92.0	127.9
	14	89.5	77.8	76.7	83.5	121.8
	16	85.8	70.6	70.6	71.2	72.4
	18	83.0	66.5	66.4	71.3	72.2
	20	99.4	63.0	62.4	71.9	75.1
	22	134.4	61.6	60.0	60.8	77.7
	24	217.0	64.9	58.8	63.5	82.0
	25	249.4	67.5	59.8	68.3	87.0
	26	306.1	73.5	61.4	69.7	91.5
	27	389.5	73.8	62.8	71.2	92.4
	28	488.1	82.4	62.3	73.6	97.1
	29	561.2	87.1	65.4	76.8	98.3
	30	672.2	93.7	69.8	75.7	103.6

All times in seconds.

Table 5: 26-node Sequent Symmetry.

VITA

Charles William Kurak Jr. has a Bachelor of Arts degree from Thomas A. Edison State College of New Jersey in Natural Sciences and Mathematics 1984 and expects to receive a Master of Science in Computer and Information Sciences from the University of North Florida, May 1991. Dr. Yap S. Chua of the University of North Florida is serving as Charles' thesis adviser. Charles has been employed as a computer programmer analyst by the United States Navy since 1984, and has been a paid Graduate Research Assistant at UNF from May, 1990 to December, 1990.

Charles' extensive professional and academic programming experience includes use of C, Pascal, and BASIC in the Unix, MS-DOS, and combined LAN environments. He has developed utilities for professional systems use including graphics, file viewing, communications, and database systems. His academic work encompasses 2-D and 3-D imaging and graphics, ray tracing, communications, systems software, data modeling, human-machine interfacing, and multi-processing.

Charles is listed in Who's Who Among Students in American Colleges and Universities, 1990. He has been a member of the IEEE and the IEEE Computer Society and the Association for Computing Machinery (ACM) since 1989. He has been serving as the Chair of the Student Chapter of the ACM at UNF (ACM-UNF) since 1989. Charles has presented his research work to the Jacksonville FL Region of the Mathematical Association of America and ACM-UNF. Charles has been married for the past twelve and one-half years. He has one son, age 11 years.