RL-TR-95-160
In-House Report
September 1995

PARALLEL IMPLEMENTATION & ANALYSIS OF LOW LEVEL VISION ALGORITHMS

Lee A. Uvanni

Rome Laboratory
Air Force Materiel Command
Griffiss Air Force Base, New York

RL-TR-95-160 has been reviewed and is approved for publication.

APPROVED: James A. Sieffert

JAMES A. SIEFFERT
Acting Chief, Image Systems Division
Intelligence and Reconnaissance Directorate

FOR THE COMMANDER: [illegible]

DELBERT B. ATKINSON, Colonel, USAF
Director of Intelligence and Reconnaissance

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE September 1995	3. REPORT TYPE AND DATES COVERED In-House 1 Nov 92 - 30 May 95

4. TITLE AND SUBTITLE PARALLEL IMPLEMENTATION & ANALYSIS OF LOW LEVEL VISION ALGORITHMS	5. FUNDING NUMBERS PR-4594 TA-18 WU-U6
6. AUTHOR(S) Lee A. Uvanni	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Rome Laboratory (IRRE) 32 Hangar Road Rome, New York 13441-4114	8. PERFORMING ORGANIZATION REPORT NUMBER RL-TR-95-160
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Laboratory (IRRE) 32 Hangar Road Rome, New York 13441-4114	10. SPONSORING/MONITORING AGENCY REPORT NUMBER N/A

11. SUPPLEMENTARY NOTES

Project Engineer: Lee A. Uvanni (315)330-4863

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release, distribution unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This in-house effort developed and implemented low-level image processing routines in parallel on any array of transputers. A selected set of low-level image processing algorithms was implemented onto a single transputer, then onto multiple transputers in two (2) different network topologies, a straight pipe configuration and a ring configuration, and implemented on multiple numbers of transputers to determine the efficiency of hardware utilization. Performance analysis includes achieved speed-up, ease of implementation, efficiency, as well as advantages and disadvantages of parallel implementation.

14. SUBJECT TERMS transputers, parallel processing, ring, pipe, image processing, hardware	15. NUMBER OF PAGES 48 16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT U/L

Table of Contents

Parallel Implementation & Analysis of Low-Level Vision Algorithms

1.0 Objective

The objective of this project was to implement low-level vision algorithms on a group of Inmos T800 transputers and perform an analysis of speed-up, performance, ease-of-implementation, efficiency of hardware utilization, and other related engineering issues.

2.0 Approach

To reach the objective, this project was broken up into several intermediate steps. The following are the general steps that were taken.

2.1 INMOS T800 Transputer

In order to fully exploit the parallel processing capabilities of the transputer, familiarization with the specifications, characteristics, and capabilities of the INMOS T800 transputer was required.

2.2 Software

Extensive knowledge of the parallel language of the transputer, Occam, was acquired. To avoid debugging problems at higher levels, simple image processing algorithms were implemented first.

2.3. Low Level Image Processing

A series of low-level image processing algorithms was implemented first sequentially, in Occam, on a single transputer. Some Occam functions were also written to assist the procedures.

2.4 Actual Parallel Implementation

Once running smoothly at the serial level, the routines were mapped onto the 17 T800 transputers, in two different configurations; the straight pipe architecture network and the ring architecture network. A comparative analysis was performed against the two networks. The algorithms were also implemented onto eight different sized mappings of transputers for each configuration to determine how efficiently the tasks were divided up for the available hardware.

2.5 Results

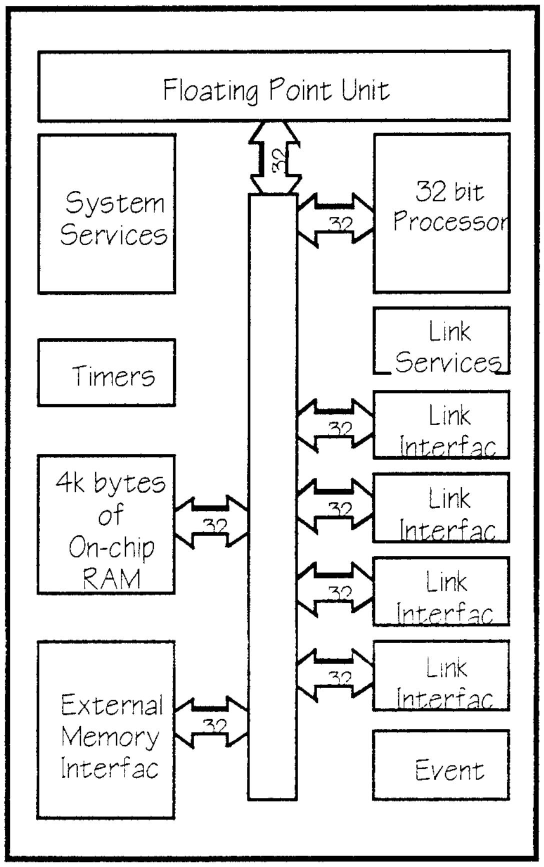
Each algorithm was executed on both networks, varying the number of transputers used from one to seventeen in increments of two. Timings were acquired for actual processing time, communication time, and the total time. Six graphs were plotted for each algorithm. The actual processing time of each algorithm for both network configurations was plotted together against the number of transputers to compare the difference in the two networks. The communication times and the total times were also plotted on the same scale for both network configurations to compare the effect communication costs have on the algorithm. Speed-Up curves for each separate network comparing processing speed-up verses actual speed-up were plotted against the number of transputers. Finally, the total efficiency against the number of processors was plotted for both networks together to compare which configuration was more efficient.

2.6 Analysis

Analysis will be broken up into two different sections. The first section will analyze the problems encountered in the project, and the second portion will analyze the results obtained from the parallel implementation. The problems encountered in the project include software problems, communication problems, and data partitioning problems. The parallel implementation analysis includes speed-up, ease-of-implementation, efficiency, and suggested improvements.

3.0 The T800 Transputer

The T800 transputer is a powerful, 32-bit CISC (Complex Instruction Set Computer) microprocessor that is capable of providing up to 10 MIPS (Millions of Instructions Per Second) of processing power. Each transputer has 1 Mbyte of on-chip memory, and four communication links that allow any number of transputers to be cascaded together in a variety of configurations. Transputers linked together can all operate in parallel, where passing data, processing data, and receiving data all are functions accomplished simultaneously, independent of each other. Seventeen T800 transputers will be utilized for actual data computations to provide parallel processing power to a series of low-level image processing algorithms. The transputers will be accessed through a VME bus connection over a SUN SPARC station. One additional transputer will be utilized to handle messages in and out of the system, and the root transputer will be used as a pass through link from the SUN host to the array of transputers. This configuration should provide a powerful parallel processing environment for this project.



FEATURES

- 32 bit architecture
- 33 ns internal clock cycle
- 30 MIPS (peak) instruction rate
- 4.3 Mflops (peak) instruction rate
- 64 bit on-chip floating point unit which conforms to IEEE 754
- 4 kbytes of on-chip RAM.
- 120 Mbytes/sec sustained data rate to internal memory
- 4 Gbytes directly addressable external memory
- 40 Mbytes/sec sustained data rate to external memory
- 630 ns response to interrupts
- Four INMOS serial links5/10/20 Mbits/sec
- Bi-directional data rate of 2.4 Mbytes/sec per link
- High performance graphics support with block move instructions
- Boot from ROM or communication links
- Single 5 MHz clock input
- Single +5V ±5% power supply
- MIL-STD-883C processing is available

Figure 1: The IMS T800 Transputer - Engineering Data[1]

When implemented in an optimal parallel manner on the transputers , these low-level image processing algorithms should run much faster, exhibiting a near-linear speed-up. This experiment provides evidence to illustrate this effect, among discussing other important issues involved in parallel processing. A thorough understanding of the selected algorithms and how they actually operate is required before they are implemented in parallel, in order to efficiently divide the algorithms up into subtasks and map the algorithms optimally onto the transputers.

[1] [1] "The Transputer Databook", pp. 189.

3.1 Processes and Concurrency

A sequence of instructions, called a process, starts, performs a number of actions, and either stops without completing or terminates complete. The transputer can run several processes concurrently, where each process is assigned a priority of either high or low. A microcoded scheduler enables any number of processes to be executed together sharing the processor time on a single transputer. The scheduler operates so that inactive processes do not consume any processing time, and active processes waiting to be executed are held into two linked lists of process workspaces. A group of transputers cascaded together can also run processes concurrently, on each transputer, where the processes can be either the same or different on each transputer. This is where the power of parallel processing is most effective.

3.2 Memory

For high data throughput rates, each T800 is equipped with 4 Kbytes of internal static memory.

3.3 Communication

A Channel is an unbuffered, unidirectional point-to-point connection for communication between two processors executing in parallel. Channels allow synchronized and unbuffered communication between processes. As a result, a channel needs no process queue, no message queue, and no message buffer. A link is defined as a pair of unidirectional channels connecting two transputers.

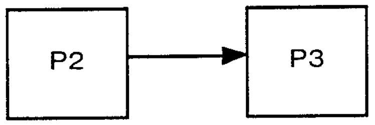


Figure 2A : One Channel

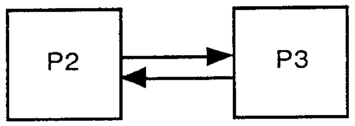


Figure 2B: Two Channels

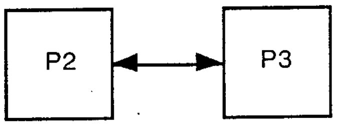


Figure 2C: One Link

Communication on the channels must be synchronized, where when the on end of a channel is ready to send or receive, the other end should be ready to do the opposite. A send process cannot proceed until the corresponding receive process on the same channel is ready. The programmer has complete control of data flow, and the route that is taken.

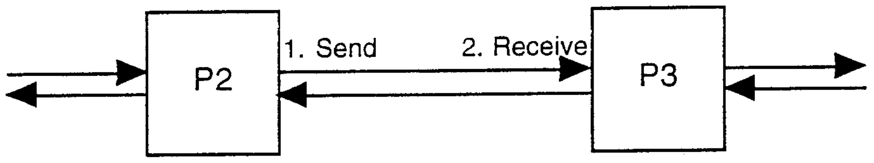


Figure 3: Proper Synchronized communication between two transputers. Processor P2 sends data to processor P3, which receives the data along the same channel.

3.4 Networks

Several transputers may be connected together to form a network, where simple link connections for point-to-point communication join the network together. The multi-transputer system is represented by a configuration of independent processors connected together by links, where concurrent processing occurs on each separate transputer. The transputer network behaves as a group of MIMD (Multiple Instruction, Multiple Data) processors. A variety of different architectural designs can be constructed to suit the need of the problem.

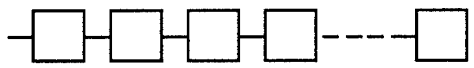


Figure 4A: Straight Pipe Architecture

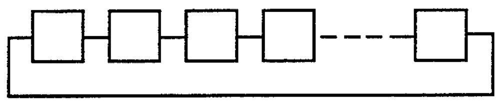


Figure 4B: Ring Architecture

4.0 Software

To fully exploit the power of the transputers, Occam, the parallel language of the transputer was used. This low-level language has proven not to be the easiest language to master. There are very few manuals out on it, and what few manuals do exist, they are very vague as to how to write routines. The two books that were mainly used, [1] & [2], gave very few examples to experiment with, which made programming much more difficult. In Occam, there are several intricate steps that must be followed before you can actually "execute" an algorithm. There are several separate files that must be set up containing important information, and they must be written painstakingly perfect.

You must have a physical understanding of how the transputers are connected together among each other in order to properly configure your different networks. First, determine the type of network configuration you will be using. Next, set up all required physical connections between transputers. Now the channels, or logical connections between transputers, must all be declared in software in a ".pgm" file. This .pgm file is the core of the programs required to execute code.

A global "include" file declares required global variables. Global variables do not mean that the value at this variable is the same on all transputers, but rather that this variable is going to be accessed on all transputers. Global information is obtained by sending data appropriately among all the transputers in the network.

Different programs must be written for the root transputer and for transputers that are not the root transputer. In the root transputer process, the root transputer must be multiplexed to be communicated to the host computer and the other transputers.

Programs must also be generated to read the data into the root transputer from the host and pass it onto the next transputer in the network. A separate program must also be written to read and pass data along the network past the root transputer.

The programs which perform actual data manipulations, such as the algorithms chosen for this project, can be generic and called by either the root transputer or any other transputer.

Occam is a very type specific language. Everything must be declared up front, and it is illegal to mix and match variables. Any mixing of types requires type casting, which is discussed further into the report.

There is no operator precedence in Occam, so parentheses define the hierarchical structure of expressions. The Occam language comes equipped with an elementary function library that provides elementary functions that are compatible with the ANSI/IEEE standard for binary floating-point arithmetic. The functions include logarithm, exponential, trigometric, inverse trigometric, polar angles, hyperbolic trigometric, and random number generation capabilities.

4.1 Serial to Parallel Conversion

The process of converting a sequential program to run on a multiprocessor seems particularly likely to result in problems of some kind. Whenever possible, the skeleton of the new program should be rewritten from scratch.

Given the great difficulty of finding bugs, especially in parallel code, much greater emphasis must be placed on writing correct code from the beginning. With this in mind, organizing programs in terms of pure mathematical functions, with clearly identified arguments and outcomes, enhances the possibility for correctness and automatic compiler checking.

Another technique to assist in the task of localizing bugs and limiting the complexity of your programs, is to use standard synchronization patterns where possible and encapsulating them as completely as possible. When all else fails in the debugging arena, an often useful approach to debugging is to sit down with pencil paper, and listing, and follow the programs manually.

5.0 Low-Level Image Processing

Image Processing required vast amounts of processing power, due to the large quantities of data involved. There are three general levels of image processing; Low-level, intermediate-level, and high-level. At the lowest level, an image is treated as raw data; a set of picture elements (pixels) without reference to the structure or objects within the image. Operations act on the image on a pixel by pixel basis in a two-dimensional grid. Data is treated like a set of features, rather than just pixels at the intermediate-levels. Higher-level routines place little emphasis on parallelism, but more on control and search strategies and knowledge about the image's domain. It is at the low-level that parallel processing is most advantageous and most efficient, since large quantities of pixels are processed simultaneously.

5.1 Selected Algorithms

The following algorithms have been implemented and use a variety of different techniques. The routines differ with the number of masks required, the number of convolutions required, the conversion of images from BYTE to INT to REAL, some required overlapping edge information, some required normalization, and some only return values as opposed to an image. All routines have been implemented on a single transputer, and on both the Straight Pipe network and the Ring network.

5.1.1 Average Pixel Intensity

This algorithm calculates and returns the value of the average pixel intensity of the input image. This algorithm requires global communication to find the average pixel intensity across the entire image. Each transputer determines the average pixel intensity of the chunk of imagery it has to process, and all the results are sent to the root transputer for a global average. No overlap, no masks, and no normalization is required, and there is no output image.

5.1.2 Threshold

This algorithm turns an eight-bit grey scale image into a single-bit black and white image, where the cut off point for black or white pixels is a predetermined threshold value. The input image and output images will be of the same size.

5.1.3 Average Value Threshold

This algorithm performs thresholding at the average intensity value of the input image. This algorithm requires global communication first to find the average pixel intensity across the entire image, and secondly to pass the information back to each transputer to threshold the image. The input image and the output image will be of the same size.

5.1.4 Generalized Convolution Filter

This algorithm is an averaging filter that convolves a specified kernel around the image, resulting in a smoothing of rough edges in the image. The kernel can range from 3x3 to 11x11.

$$Fs(x,y) = [f(x,y)] * \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

where Fs(x,y) is the new image value at (x,y) and "*" implies convolution.

This input image requires a variable sized mask (3x3 to 11x11), and also requires overlapping edge information. The resulting image can be locally normalized by dividing the output by the sum of the mask. The resulting image will be half the size of the mask in rows and columns smaller on the outer perimeter edges due to the overlap.

5.1.5 Four Nearest Neighbor Mean Filter

A smoothing filter that calculates the mean value of each pixel and its four nearest neighbors. The output image is one row and one column smaller along the perimeter due to the lacking overlap information at the perimeter.

5.1.6 Eight Nearest Neighbor Mean Filter

A smoothing filter that calculates the mean value of each pixel and its eight nearest neighbors. The output image is one row and one column smaller along the perimeter due to the lacking overlap information at the perimeter.

5.1.7 Four Nearest Neighbor Median Filter

A smoothing filter that calculates the median value of each pixel in an image and its four nearest neighbors. The output image is one row and one column smaller along the perimeter due to the lacking overlap information at the perimeter.

5.1.8 Eight Nearest Neighbor Median Filter

A smoothing filter that calculates the median value of each pixel in an image and its eight nearest neighbors. The output image is one row and one column smaller along the perimeter due to the lacking overlap information at the perimeter.

5.1.9 Sobel Edge Filter

This edge detection routine calculates the sum of two separate convolutions using two separate 3x3 kernels for its results.

$$Fs(x,y) = 0.125 \times \left\{ \left[f(x,y) * \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \right] + \left[f(x,y) * \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \right] \right\}$$

where Fs(x,y) is the Sobel value at (x,y) and "*" implies convolution.

This input image requires two 3x3 sized masks, overlapping edge information, and also requires global normalization. This routine cannot be locally normalized by dividing by the sum of the masks, because both sums added up to zero, making division impossible. The resulting image will be one row and one column smaller on each of the outer perimeter edges due to the overlap.

5.1.10 Roberts Cross Edge Detection Filter

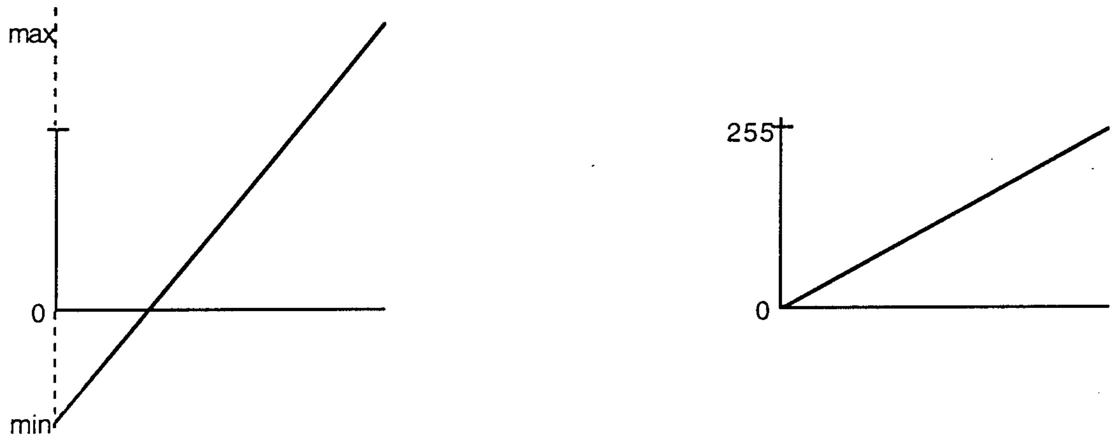
This edge detection filter outputs the maximum of the diagonal difference over every 2x2 window of the image.

$$\begin{array}{cc} a & b \\ c & d \end{array} \quad \text{--->} \quad \max(|a-d|,|b-c|)$$

This image does not require a mask, but does require an overlap of one in the right and downward directions. Although one can see from the above formula that no resulting pixel could be negative, nor could they be larger than the input image, the output image requires normalization to spread out the range of pixel values from a small gap to a standard range from 0 to 255. The resulting image will be one row and one column smaller on only two outer perimeter edges (the right edge, and the bottom edge) due to the overlap.

5.1.11 Normalization

This algorithm converts or "normalizes" the values of an image ranging from the most negative number to the most positive number, to an image that ranges from 0 to 255. The input image and the output image are the same size. This algorithm was not directed executed, but instead was utilized by several of the above algorithms internally.



5.2 Functions

Four simple functions have been implemented simply to reduce the amount of code in the algorithms, and to make the programs easier to follow by avoiding the use of a series of IF-THEN statements.

5.2.1 Minvalue

This function returns the minimum value of two values passed to the routine.

5.2.2 Maxvalue

This function returns the maximum value of two values passed to the routine.

5.2.3 Abs

This function returns the absolute value of an integer passed to the routine.

5.2.4 Sorter

This function sorts an array into ascending order using a modified bubble sort. Instead of comparing n data items O(n) times, it compares them < O(n). There is a flag that is set if a complete comparison of the n data items yields no switches, then the sorter is finished, and does not have to keep checking already sorted elements.

6.0 Actual Parallel Implementation

This was the most difficult portion of the project. There are really two aspects to this portion of the problem. First, Hardware configuration. The transputers must first be properly connected physically, and also configured in software to match the chosen hardware design. This takes a lot of concentration to get correct. For the multiple transputer configurations, a straight pipe network and a ring network have been implemented. The straight pipe can only pass data in one direction from the root, where the ring can pass data in two different directions from the root simultaneously. This bidirectional method of concurrent data passing through the transputers should cut down on communication time.

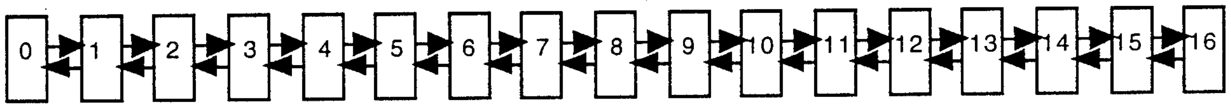


Figure 5: The Straight Pipe Network

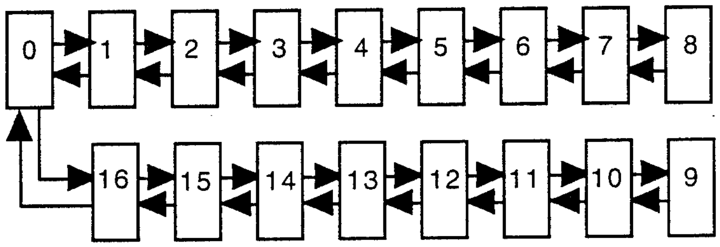
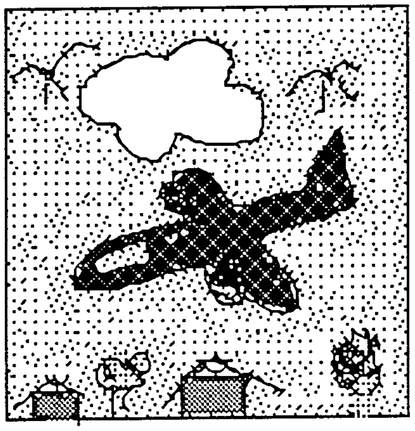


Figure 6: The Ring Network

6.1 Data

The data used for this project consisted of strictly 256x256 sized imagery, 8 bits deep providing 256 shades of grey. Any sized imagery could be utilized with this software, but a small representation of the image was chosen since so many different algorithms had to be executed in so many different configurations. All of the masks, or kernels, used in the algorithms were 3x3 pixels, although the software written can also support any size mask desired.



a	b	c
d	e	f
g	h	i

Figure 7A: Sample Image, Size 256 x 256 Pixels

Figure 7B: Sample Mask, size 3x3 pixels.

6.2 Program Effectiveness

To effectively take advantage of parallel processing, the number of process creations and synchronizations in a partitioned algorithm must be minimized. Since process creations are more expensive than process synchronizations, the usual tactic is to create the desired number of processes when the algorithm begins execution and to synchronize them when necessary. Communication overhead will be significant if synchronization is done frequently. Because synchronizations are so expensive, a goal of the parallel algorithm designer should be to make the grain size, or relative amount of work done between synchronizations, as large as possible, while keeping all the processors busy.[2]

6.3 Partitioning The Data

Partitioning is the sharing of a computation where a problem is divided into subproblems that are shared by individual processors and the solutions to the subproblems are combined to form the problem solution. Processing tasks should be subdivided to make the most effective use of available parallel hardware. Since the algorithms selected do not require more than one actual process than can be accomplished at a time, the data will be partitioned among the transputers instead of partitioning the algorithms. The most difficult portion of coding was writing the code to partition the imagery into proper sized chunks to send to the next transputer in the network,

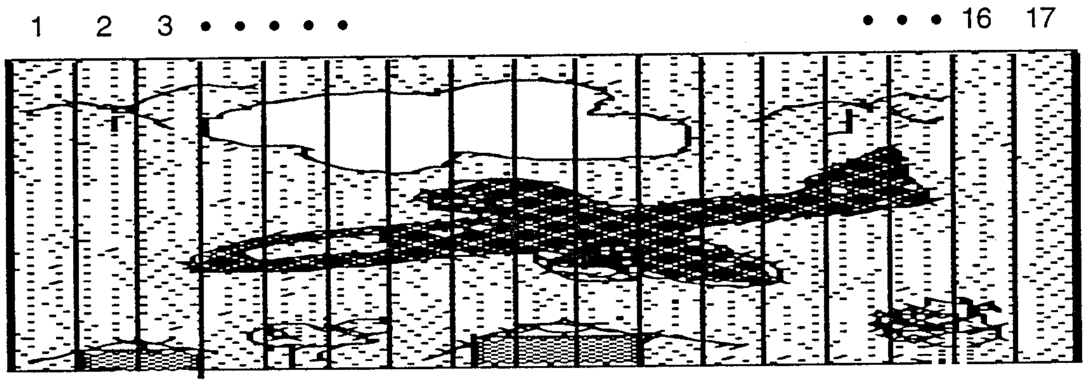


Figure 8: The 256 x 256 sized image broken up into approximately equal sections for processing among 17 transputers. Two sections have one extra column since 17 divides into 256 15 times, with a remainder of 2. The image has been distorted here for visualization purposes.

[2] [2] Designing Efficient Algorithms, page61.

what data to keep, and what data to pass on to the next one. The image must be divided into approximately equal portions for each transputer, in chunks of columns. If the image does not divide up equally, an extra column must be given to the first transputers in the network until all the extra columns are used up.

6.4 Edge Effects

Although an image may divide equally among processors, the overlapping edges of the data on each transputer must also be taken into consideration to achieve correct results at the edges. The outer perimeter of the image must also be taken into consideration, where there is no data to compensate for the overlap. The output image may be smaller than the input image at the outer perimeter in some algorithms, so it with be buffered with zeros to avoid erroneous results in the output image.

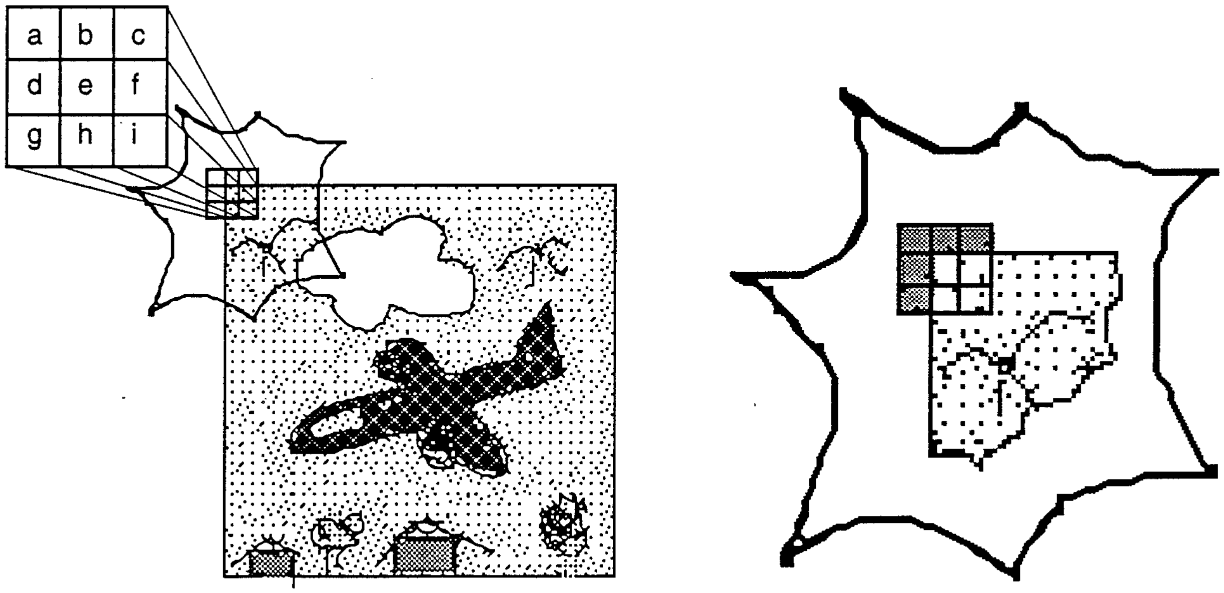


Figure 9: When the mask is passed along the image, we can see there is an overlap created equal to one half the size of the mask being used on the image (truncated one half). In this case, the 3x3 sized mask creates an overlap of 1 around the image.

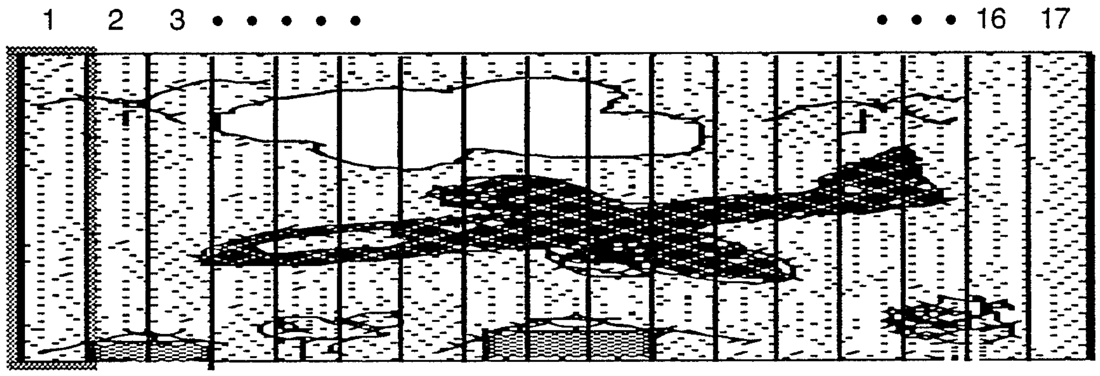


Figure 10: The overlap created by the mask requires an extra column on both sides of the image chunk in order to process the entire image correctly, which is illustrated by the shaded portion. The perimeter along the top and the bottom of the image chunk, as well as the left and right sides of the complete image, are compensated by filling the output image with zeros at these edges.

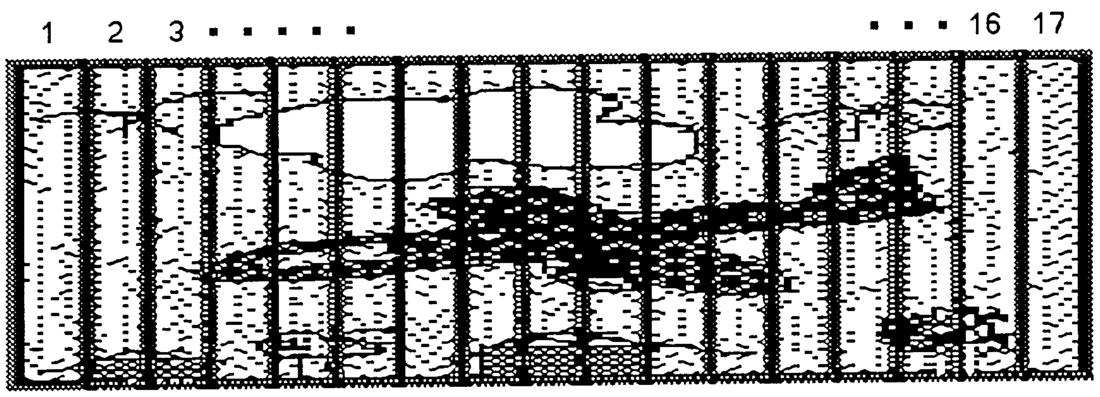


Figure 11: Each chunk of imagery requires this overlap, creating an extra two columns of imagery required on every transputer.

For illustration purposes, the seventeen chunks of data will be referred to in order from zero to sixteen, rather than from one to seventeen. This will be easier to visualize which transputer is processing which portion of the image. Each transputer will process the same chunk of data, regardless of being in the straight pipe or ring configuration, when in the multiple transputer mode. The difference will be how the data gets to each transputer, and the amount of time it takes. When executing algorithms on a single transputer, the entire image will be processed on the root transputer, where the only edge effects necessary for consideration are the four outer perimeter edges of the image.

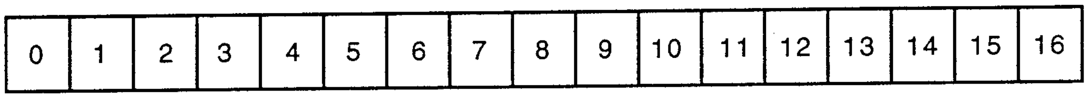


Figure 12: A representation of the 256 x 256 pixel image divided into its 17 chunks, ranging from 0 to 16.

6.5 Naming Convention

Determining what portion of data gets passed where in the network can become quite confusing, especially if data is flowing simultaneously in two directions as in the case of the ring network. For this purpose, some sort of naming convention must be established to limit the amount of confusion.

6.5.1 Straight Pipe Network

For the straight pipe, data can only flow up the pipe or down the pipe. Data flowing from a lower numbered transputer to a higher numbered transputer, as in the case of Figure 15, is data flowing UP the pipe. Data flowing from a higher numbered transputer to a lowered one is data flowing DOWN the pipe.

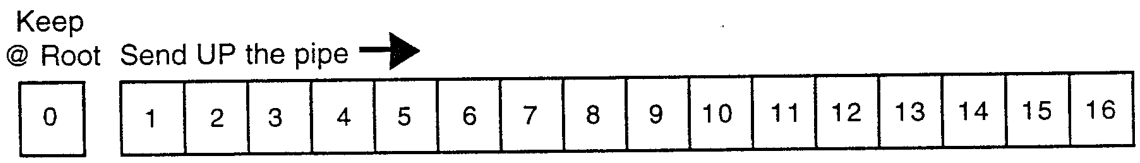


Figure 13: A simplified view of data flow using the seventeen transputer pipe configuration. The root transputer keeps the first portion of the image, and passes the rest of the data up the pipe as indicated.

6.5.2 Ring Network

For the ring network, data is flowing in four directions, so a naming convention is a little more complicated here. If the ring can be visualized as transputers #1-8 on top of the ring, and transputers #9-16 on the bottom of the ring, the ring can now have an upper and lower portion to it. Data flowing from a lower numbered transputer to a higher numbered transputer on the upper portion of the ring is data flowing UP the upper portion, and the reverse is data flowing DOWN the upper portion. Data flowing from a higher numbered transputer to a lower numbered transputer on the lower portion of the ring is data flowing UP the lower portion, and the reverse is data flowing DOWN the lower portion. The data in Figure 16 is flowing UP both portions of the ring network, and should make this explanation easier to understand.

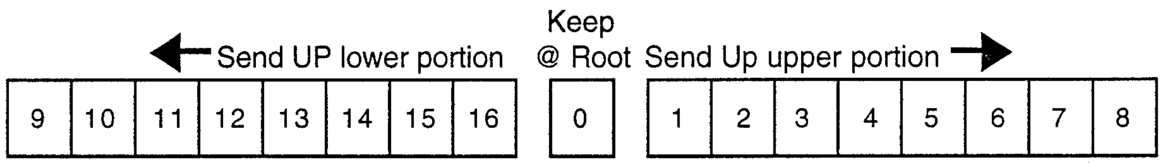


Figure 14: A simplified view of data flow using the seventeen transputer ring configuration. The root transputer keeps the first portion of the image, and passes half of the remaining data up the upper portion of the ring, and the other half up the lower portion of the ring.

6.6 Distributing Data

Although each transputer obtains the exact portion of data, regardless of the network configuration it is in, the manner in which the data is distributed is quite different among configurations.

6.6.1 Pipe Configuration

As the data is passed up the pipe, each transputer in the ring keeps the designated amount data it requires, and passes the rest on to the next transputer in the ring. This trickling effect is continued until the last transputer in the pipe is reached. At this point all the appropriate data is contained in all of the corresponding transputers. To return the data back to the root transputer, the data has to trickle down the pipe from transputer #16 all the way back down to the root transputer, since the root transputer is the only transputer that can read and write to the system. As the data comes down the pipe it is pieced together at each transputer, combining any data already received with its own data. Once all the data reaches the root transputer, the new processed image is written out to a file.

6.6.2 Ring Configuration

As the data is passed up the pipe, each transputer in the ring keeps the designated amount of data it requires, and passes the rest on to the next transputer in the ring. This trickling effect is continued until transputer #8 up the pipe, and transputer #9 down the pipe are reached. Once these two transputers are reached, all the appropriate data will be contained in all of the corresponding transputers. Although transputers #8 and #9 are physically connected within the ring, there is no need to pass data to or from these two in this configuration. To return the data back to the root transputer, the data has to trickle down the upper and lower portions of the ring simultaneously. Transputer #8 starts to send data down the upper portion of the ring, while transputer #9 starts to send data down simultaneously, where each transputer is combining receiving data with its own along the way before passing it on. At the root transputer, the root must combine both the upper portion of the image with its own data, and also the lower portion of the image before the new processed image is written out to a file. Once this is all completed, the image can be processed in any manner necessary. Then, the processed results must be sent back in the proper order to put together the entire processed image.

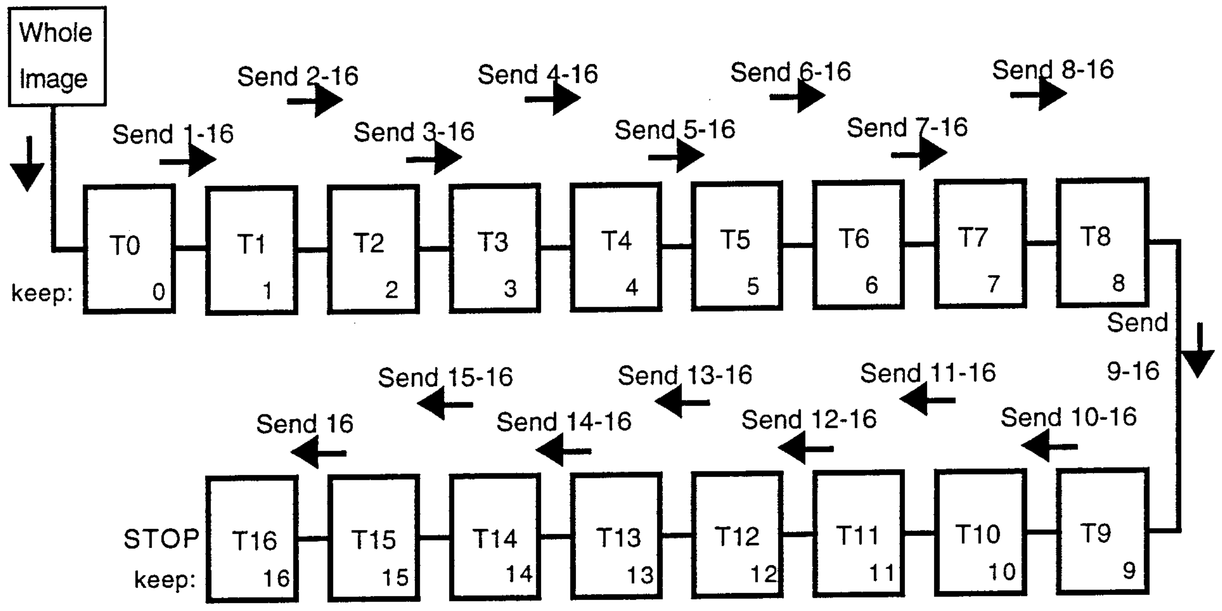


Figure 15: The data trickling effect UP the pipe of transputers.

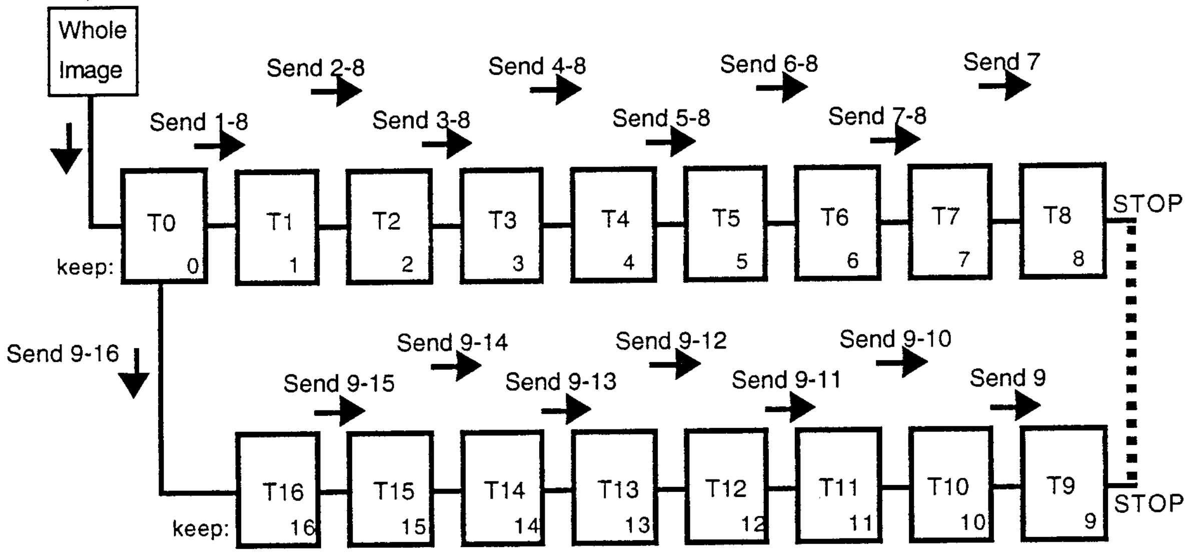


Figure 16: The data trickling effect Up both portions of the ring of transputers simultaneously.

To summarize, the mask and imagery must be read in, the required overlap must be determined, the size of the image chunks must be calculated, any excess portion of the image that needs to be distributed must be calculated, whatever portion of the image the root transputer does not require must be passed on to the other transputers, each transputer in the network must save necessary data and pass unnecessary data to subsequent transputers in the ring, calculate all the results, calculate required timings, pass the data back to the root transputer combining it in the proper order, then write the new image file to memory.

6.7 Communication vs. Processing

Since communication costs must be considered when determining the complexity of a parallel algorithm, a convention must be established for determining what constitutes communication, and what actually constitutes actual data computations, or actual processing. The clocks on each separate transputer are not synchronized, so the time recorded on one transputer has no meaningful relationship to the time recorded on a different transputer. The only significant information from the clocks is the difference between two separate readings of the clock on the same transputer. So if the starting time is taken from the root transputer, and the finishing time is taken from the last transputer in the network, the difference in these two numbers will be irrelevant. For this reason, the timing calculations will be done only on two transputers, the root transputer and the first transputer connected directly to the root. The root transputer will record the starting and ending times of the entire algorithm, since all algorithms must start and finish on the root transputer, regardless of the network configuration. All actual computation times will be taken from the first transputer. Since all transputers are performing the same computations and are doing approximately the same amount of work, they should all require approximately the same amount of time to perform. Since the first transputer is connected directly to the root transputer, and data needs to be passed to the root transputer to perform all additional calculations, this is the easiest processor to use because it can pass the data directly to the root without having to trickle down the network from anywhere else. So the timing convention will be as follows: Total time will be defined as the time it requires from start to finish to perform a given process on all transputers in the network. Computation time will be the time required for a single transputer to perform all necessary calculations in the process. I/O time will be the Total Time minus the Computation Time.

7.0 Results

The next twenty pages illustrate the results of the ten selected algorithms using both the pipe and the ring networks, for one through seventeen transputers in increments of two. Six graphs are presented for a better understanding of how effective each algorithm performed. The first three graphs are actual data results, while the last three required some calculations of these results. The speedup achieved by a parallel algorithm running on p processors is the ratio between the time taken by that parallel computer executing the fastest serial algorithm and the time taken by the same parallel computer executing the parallel algorithm on p processors.[3] The computational speed-up is the speed-up considering only the time it took to perform the computations of the algorithms ignoring communication, where the total speed-up is the speed-up of the total time including computation and communication. The efficiency of a parallel algorithm running on p processors is the speed-up divided by p.[4] Only the total efficiency will be taken into consideration for this project.

[3] [2] Designing Efficient Algorithms, pp. 43.

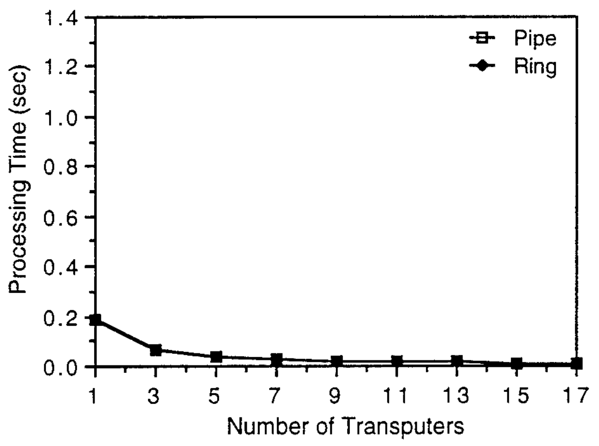
[4] [2] Designing Efficient Algorithms, pp. 43.

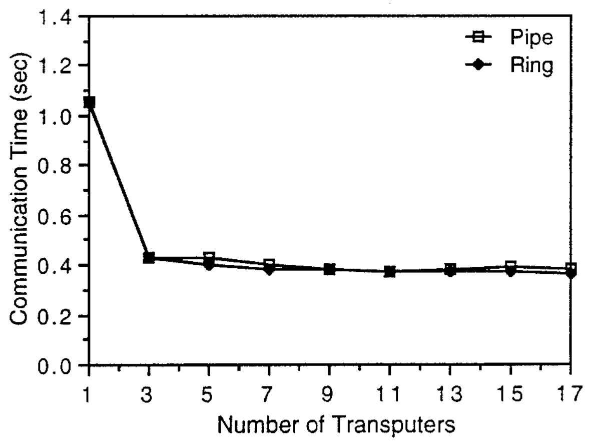
Table 1A: Average Pixel Intensity Value (PIPE) Results

#T800s	Tp(sec)	Tc(sec)	Tt(sec)	Computational Speed Up	Total Speed Up	Efficiency
1	0.189	1.052	1.241	1.00	1.00	1.00
3	0.062	0.433	0.495	3.05	2.51	0.84
5	0.037	0.427	0.464	5.11	2.67	0.53
7	0.026	0.398	0.425	7.27	2.92	0.42
9	0.021	0.384	0.405	9.00	3.06	0.34
11	0.017	0.378	0.395	11.12	3.14	0.29
13	0.014	0.385	0.399	13.50	3.11	0.24
15	0.012	0.392	0.404	15.75	3.07	0.20
17	0.011	0.385	0.396	17.18	3.13	0.18

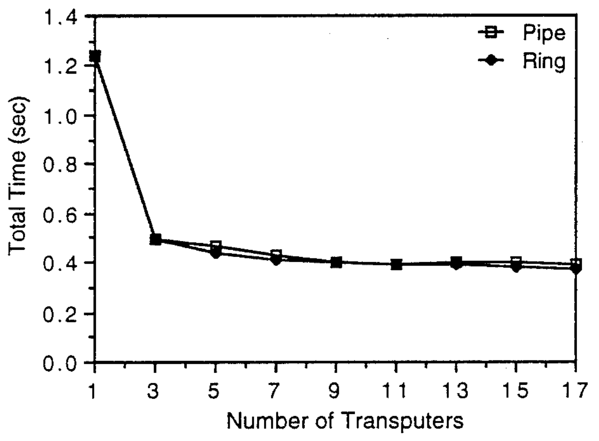
Table 1B: Average Pixel Intensity Value (RING) Results

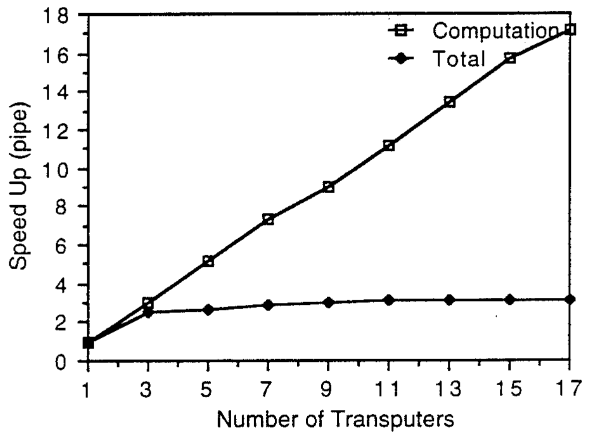
#T800s	Tp(sec)	Tc(sec)	Tt(sec)	Computational Speed Up	Total Speed Up	Efficiency
1	0.189	1.052	1.241	1.00	1.00	1.00
3	0.064	0.427	0.491	2.95	2.53	0.84
5	0.038	0.399	0.437	4.97	2.84	0.57
7	0.027	0.385	0.413	7.00	3.00	0.43
9	0.021	0.380	0.402	9.00	3.09	0.34
11	0.018	0.376	0.394	10.50	3.15	0.29
13	0.015	0.374	0.389	12.60	3.19	0.25
15	0.013	0.374	0.387	14.54	3.21	0.21
17	0.011	0.367	0.378	17.18	3.28	0.19



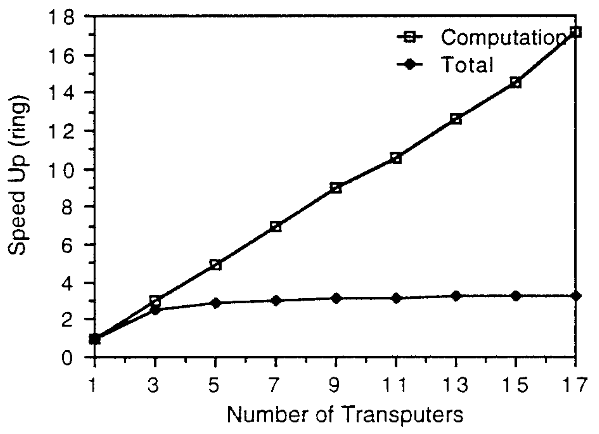


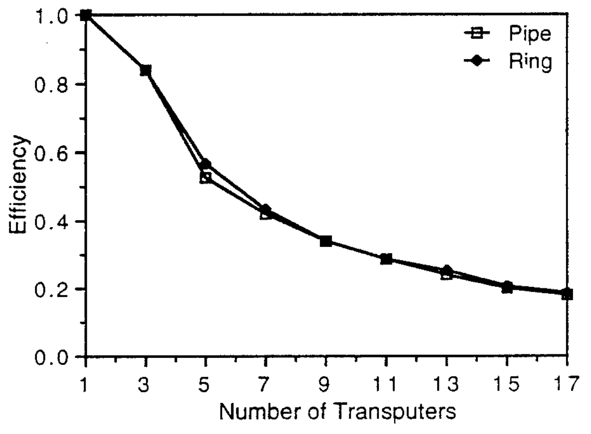
Figures 17A-17B: Figure 17A: Processing time curves for the pipe and ring architectures for the average pixel intensity value algorithm. Figure 17B: Communication time curves for the same algorithm for both architectures.





Figures 17C-17D: Figure 17C: Total time curves for the pipe and ring architectures. Figure 17D: Speed Up curves for the algorithm using the pipe architecture.





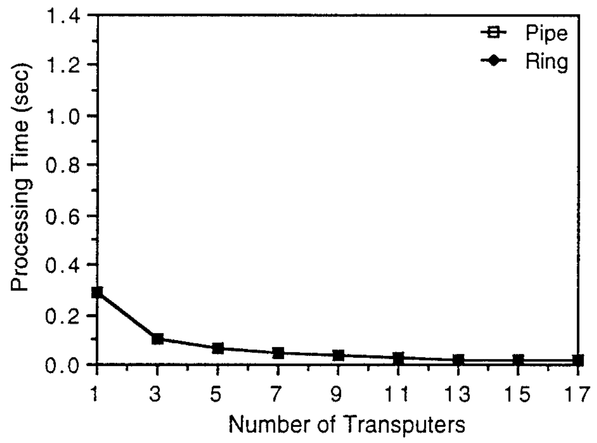
Figures 17E-17F: Figure 17E: Speed Up curves for the algorithm using the ring architecture. Figure 17F: Efficiency of the pipe and ring architectures using the total algorithm time.

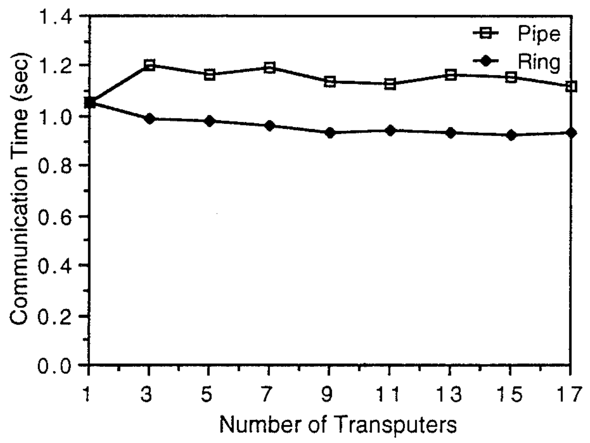
Table 2A: Threshold @ 128 Algorithm (PIPE) Results

#T800s	Tp(sec)	Tc(sec)	Tt(sec)	Computational Speed Up	Total Speed Up	Efficiency
1	0.294	1.057	1.351	1.00	1.00	1.00
3	0.102	1.208	1.310	2.88	1.03	0.34
5	0.061	1.167	1.228	4.82	1.10	0.22
7	0.043	1.199	1.242	6.84	1.09	0.16
9	0.034	1.135	1.169	8.65	1.16	0.13
11	0.028	1.130	1.158	10.50	1.17	0.11
13	0.023	1.171	1.195	12.25	1.13	0.09
15	0.020	1.159	1.179	14.70	1.15	0.08
17	0.018	1.123	1.141	16.33	1.18	0.07

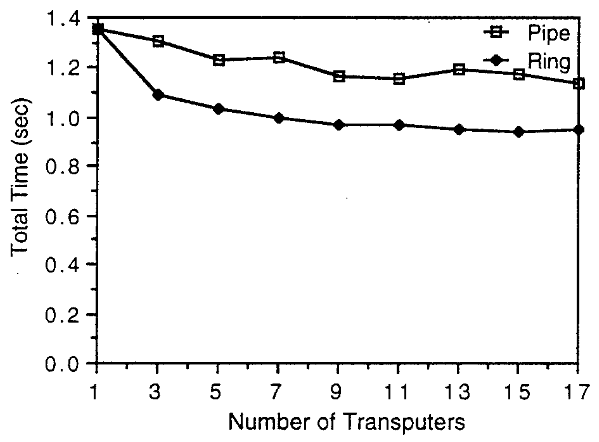
Table 2B: Threshold @ 128 Algorithm (RING) Results

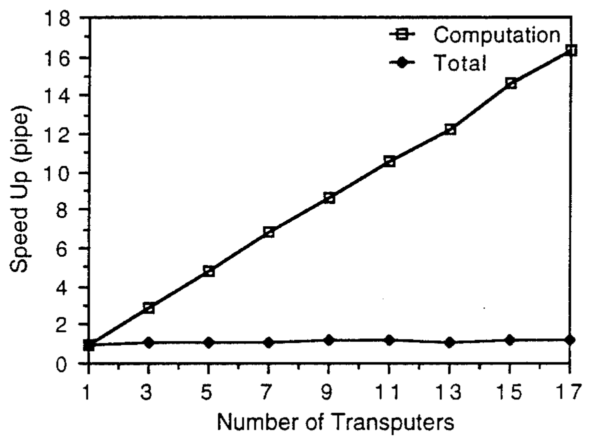
#T800s	Tp(sec)	Tc(sec)	Tt(sec)	Computational Speed Up	Total Speed Up	Efficiency
1	0.294	1.057	1.351	1.00	1.00	1.00
3	0.102	0.991	1.093	2.88	1.03	0.34
5	0.061	0.976	1.037	4.82	1.10	0.22
7	0.043	0.959	1.002	6.84	1.09	0.16
9	0.034	0.936	0.970	8.65	1.16	0.13
11	0.028	0.945	0.973	10.50	1.17	0.11
13	0.023	0.930	0.954	12.25	1.13	0.09
15	0.020	0.926	0.946	14.70	1.15	0.08
17	0.018	0.932	0.950	16.33	1.18	0.07



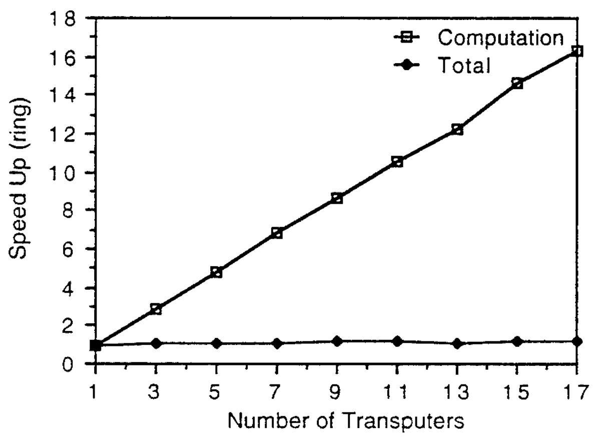


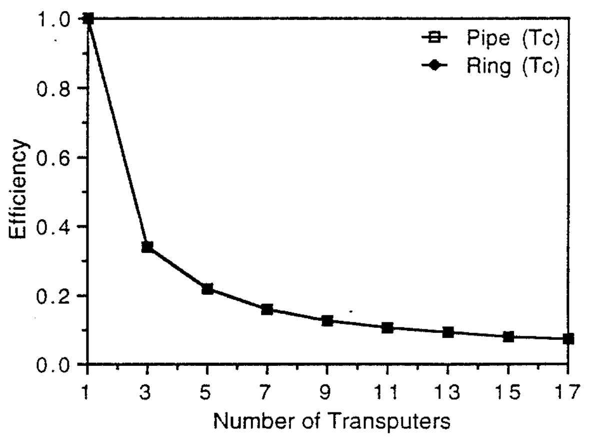
Figures 18A-18B: Figure 18A: Processing time curves for the pipe and ring architectures for the threshold algorithm. Figure 18B: Communication time curves for the same algorithm for both architectures.





Figures 18C-18D: Figure 18C: Total time curves for the pipe and ring architectures. Figure 18D: Speed Up curves for the algorithm using the pipe architecture.





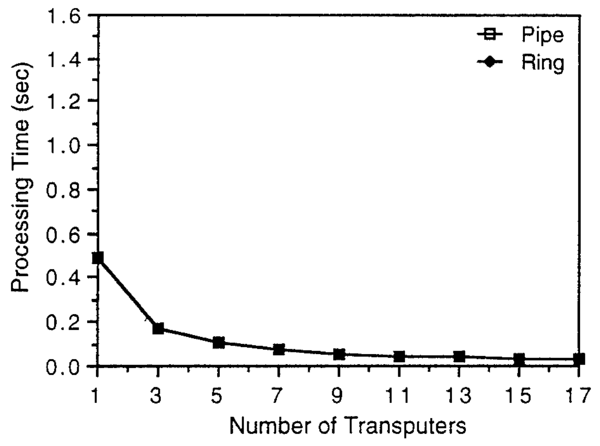
Figures 18E-18F: Figure 18E: Speed Up curves for the algorithm using the ring architecture. Figure 18F: Efficiency of the pipe and ring architectures using the total algorithm time.

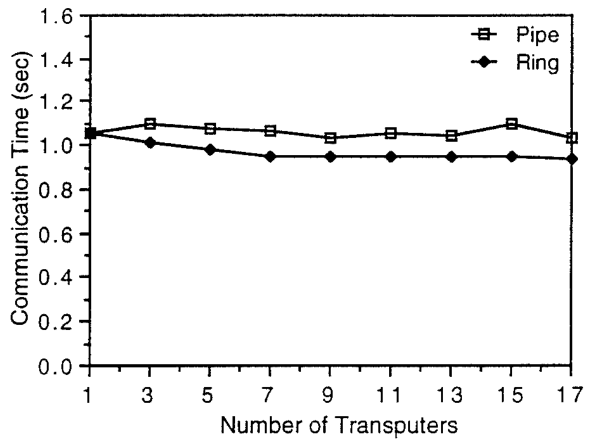
Table 3A: Threshold @ Average Pixel Intensity Algorithm (PIPE) Results

#T800s	Tp(sec)	Tc(sec)	Tt(sec)	Computational Speed Up	Total Speed Up	Efficiency
1	0.491	1.051	1.542	1.00	1.00	1.00
3	0.169	1.098	1.267	2.91	1.22	0.41
5	0.102	1.075	1.177	4.81	1.31	0.26
7	0.072	1.062	1.134	6.82	1.36	0.19
9	0.057	1.040	1.083	8.61	1.42	0.16
11	0.047	1.052	1.099	10.45	1.40	0.13
13	0.039	1.049	1.088	12.59	1.42	0.11
15	0.33	1.101	1.134	14.88	1.36	0.09
17	0.029	1.038	1.067	16.93	1.45	0.08

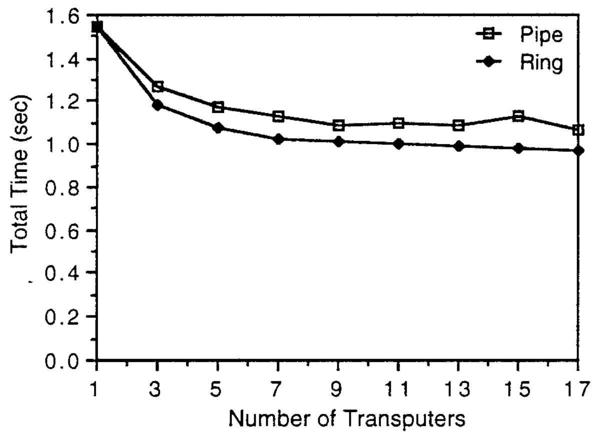
Table 3B: Threshold @ Average Pixel Intensity Algorithm (RING) Results

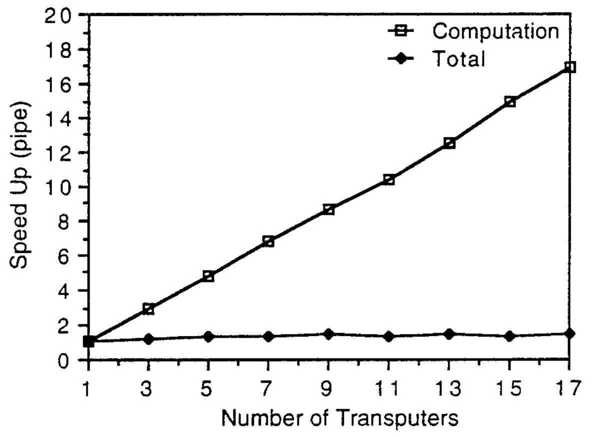
#T800s	Tp(sec)	Tc(sec)	Tt(sec)	Computational Speed Up	Total Speed Up	Efficiency
1	0.491	1.051	1.542	1.00	1.00	1.00
3	0.169	1.017	1.187	2.91	1.42	0.47
5	0.102	0.980	1.082	4.81	1.43	0.29
7	0.072	0.953	1.025	6.82	1.50	0.21
9	0.057	0.954	1.011	8.61	1.49	0.17
11	0.047	0.951	0.998	10.45	1.55	0.14
13	0.39	0.947	0.987	12.59	1.56	0.12
15	0.033	0.946	0.983	14.88	1.57	0.10
17	0.029	0.943	0.972	16.93	1.59	0.09



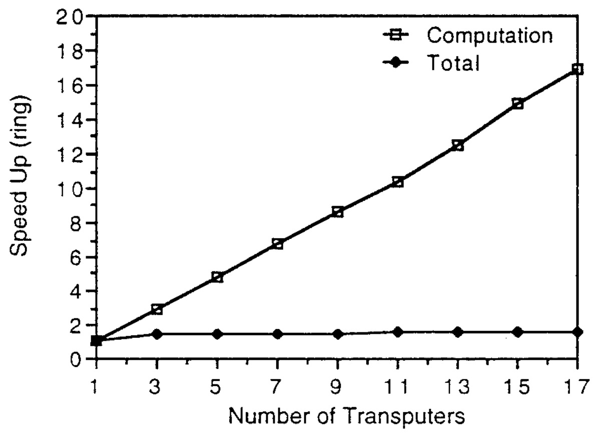


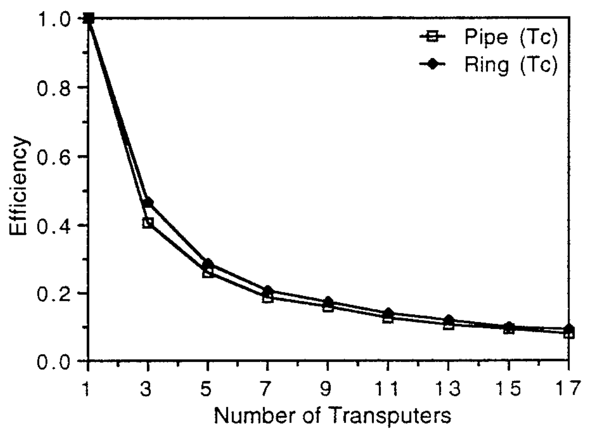
Figures 19A-19B: Figure 19A: Processing time curves for the pipe and ring architectures for the threshold at the average intensity value algorithm. Figure 19B: Communication time curves for the same algorithm for both architectures.





Figures 19C-19D: Figure 19C: Total time curves for the pipe and ring architectures. Figure 19D: Speed Up curves for the algorithm using the pipe architecture.





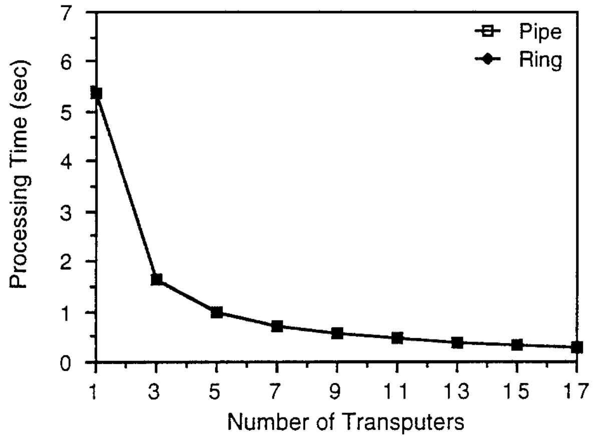
Figures 19E-19F: Figure 19E: Speed Up curves for the algorithm using the ring architecture. Figure 19F: Efficiency of the pipe and ring architectures using the total algorithm time.

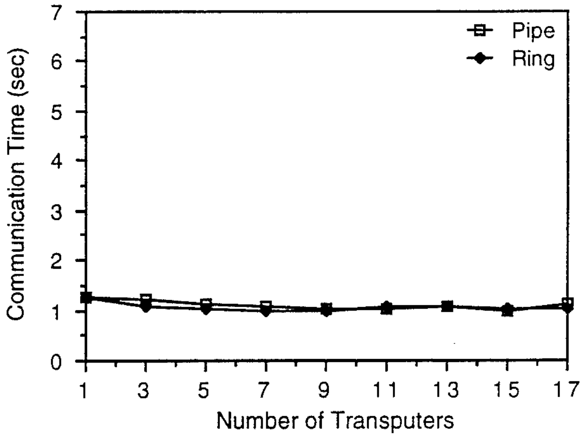
Table 4A: 3x3 Convolution Filter Algorithm (PIPE) Results

#T800s	Tp(sec)	Tc(sec)	Tt(sec)	Computational Speed Up	Total Speed Up	Efficiency
1	5.345	1.254	6.599	1.00	1.00	1.00
3	1.637	1.203	2.840	3.27	2.32	0.77
5	0.982	1.142	2.124	5.44	3.11	0.62
7	0.708	1.057	1.765	7.55	3.74	0.53
9	0.555	1.015	1.570	9.63	4.20	0.47
11	0.459	1.048	1.507	11.64	4.38	0.40
13	0.383	1.071	1.454	13.96	4.54	0.35
15	0.325	1.000	1.325	16.45	4.95	0.33
17	0.287	1.136	1.423	18.62	4.64	0.27

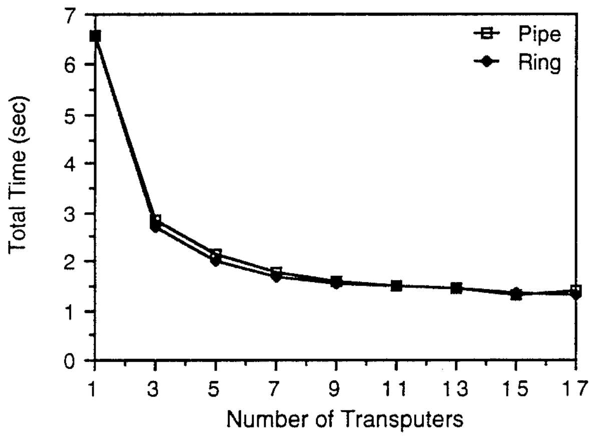
Table 4B: 3x3 Convolution Filter Algorithm (RING) Results

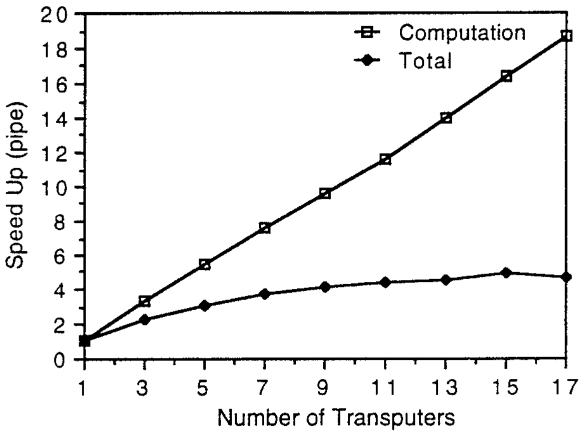
#T800s	Tp(sec)	Tc(sec)	Tt(sec)	Computational Speed Up	Total Speed Up	Efficiency
1	5.345	1.254	6.599	1.00	1.00	1.00
3	1.613	1.084	2.697	3.31	2.45	0.82
5	0.968	1.037	2.006	5.52	3.29	0.66
7	0.698	0.979	1.677	7.66	3.94	0.56
9	0.547	0.977	1.524	9.77	4.33	0.48
11	0.453	1.053	1.506	11.80	4.38	0.40
13	0.377	1.051	1.428	14.18	4.62	0.36
15	0.321	1.021	1.342	16.65	4.92	0.33
17	0.283	1.021	1.304	18.89	5.06	0.30



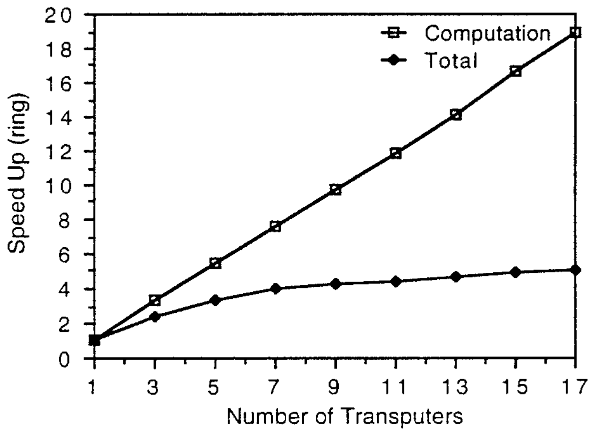


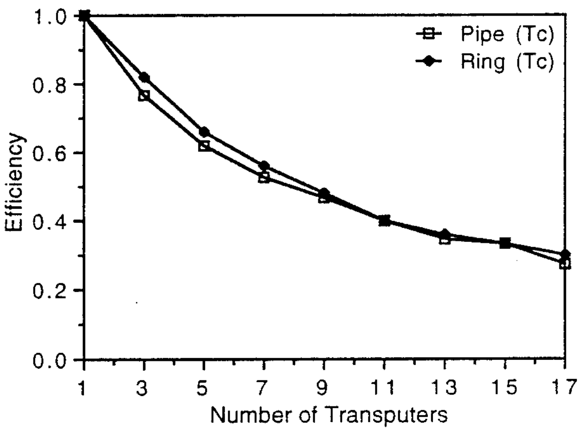
Figures 20A-20B: Figure 20A: Processing time curves for the pipe and ring architectures for the 3x3 convolution algorithm. Figure 20B: Communication time curves for the same algorithm for both architectures.





Figures 20C-20D: Figure 20C: Total time curves for the pipe and ring architectures. Figure 20D: Speed Up curves for the algorithm using the pipe architecture.





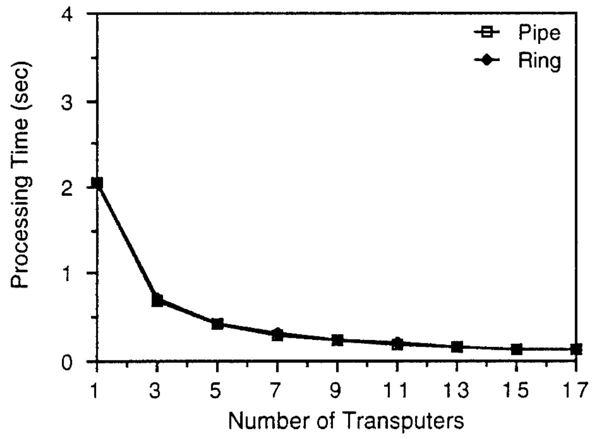
Figures 20E-20F: Figure 20E: Speed Up curves for the algorithm using the ring architecture. Figure 20F: Efficiency of the pipe and ring architectures using the total algorithm time.

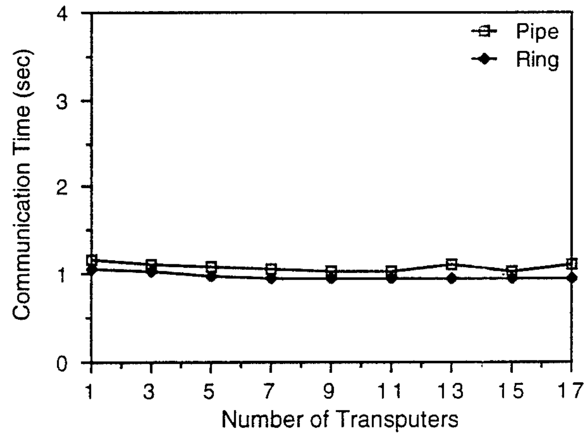
Table 5A: Four Nearest Neighbor Mean Filter Algorithm (PIPE) Results

#T800s	Tp(sec)	Tc(sec)	Tt(sec)	Computational Speed Up	Total Speed Up	Efficiency
1	2.057	1.156	3.109	1.00	1.00	1.00
3	0.704	1.101	1.805	2.92	1.72	0.57
5	0.423	1.084	1.507	4.86	2.06	0.41
7	0.305	1.060	1.365	6.74	2.28	0.33
9	0.239	1.042	1.281	8.61	2.43	0.27
11	0.198	1.038	1.236	10.39	2.52	0.23
13	0.165	1.120	1.285	12.47	2.42	0.19
15	0.141	1.043	1.183	14.59	2.63	0.17
17	0.124	1.102	1.226	16.59	2.54	0.14

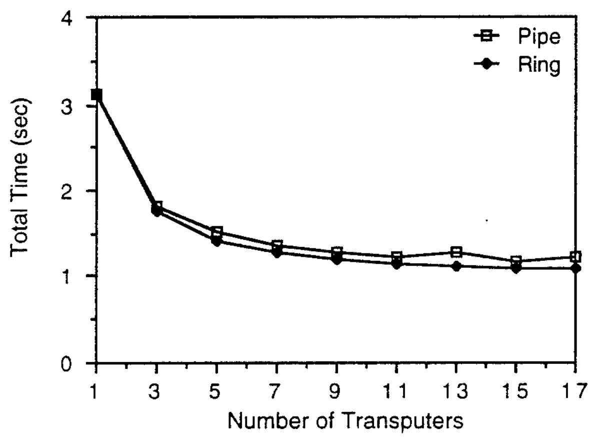
Table 5B: Four Nearest Neighbor Mean Filter Algorithm (RING) Results

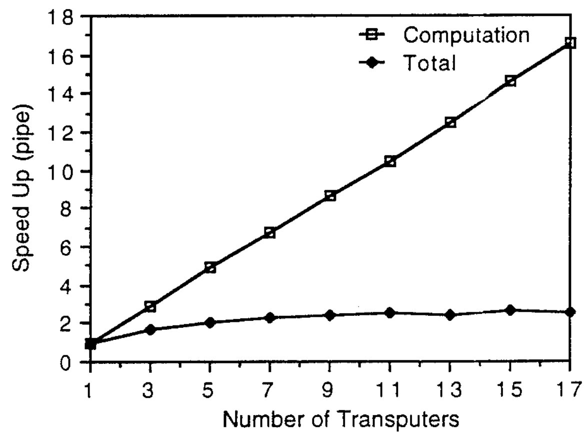
#T800s	Tp(sec)	Tc(sec)	Tt(sec)	Computational Speed Up	Total Speed Up	Efficiency
1	2.057	1.156	3.109	1.00	1.00	1.00
3	0.713	1.046	1.760	2.89	1.77	0.59
5	0.428	0.980	1.408	4.81	2.21	0.44
7	0.308	0.965	1.272	6.68	2.44	0.35
9	0.241	0.956	1.198	8.54	2.60	0.29
11	0.200	0.947	1.147	10.29	2.71	0.25
13	0.167	0.943	1.110	12.32	2.80	0.22
15	0.142	0.948	1.090	14.49	2.85	0.19
17	0.125	0.957	1.082	16.46	2.87	0.17



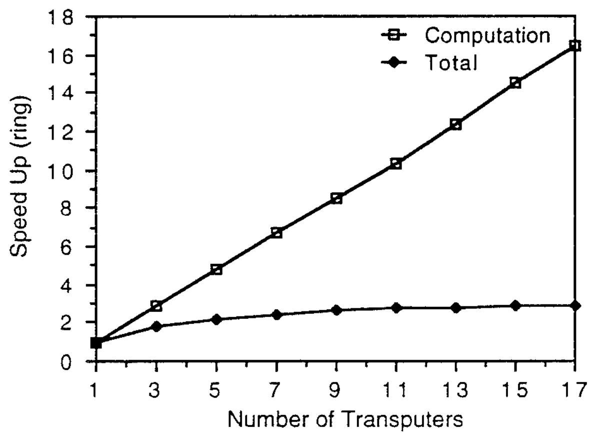


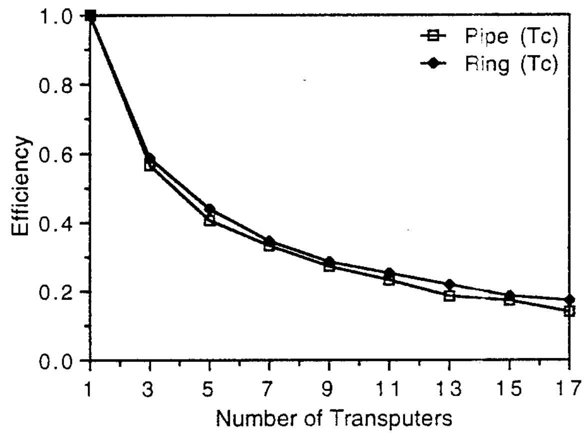
Figures 21A-21B: Figure 21A: Processing time curves for the pipe and ring architectures for the four nearest neighbor mean algorithm. Figure 21B: Communication time curves for the same algorithm for both architectures.





Figures 21C-21D: Figure 21C: Total time curves for the pipe and ring architectures. Figure 21D: Speed Up curves for the algorithm using the pipe architecture.





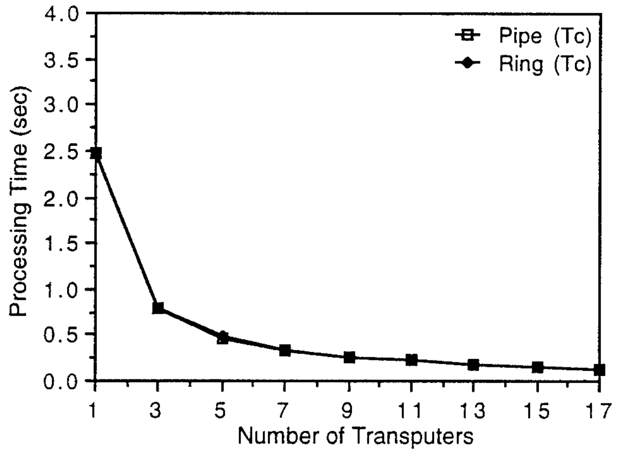
Figures 21E-21F: Figure 21E: Speed Up curves for the algorithm using the ring architecture. Figure 21F: Efficiency of the pipe and ring architectures using the total algorithm time.

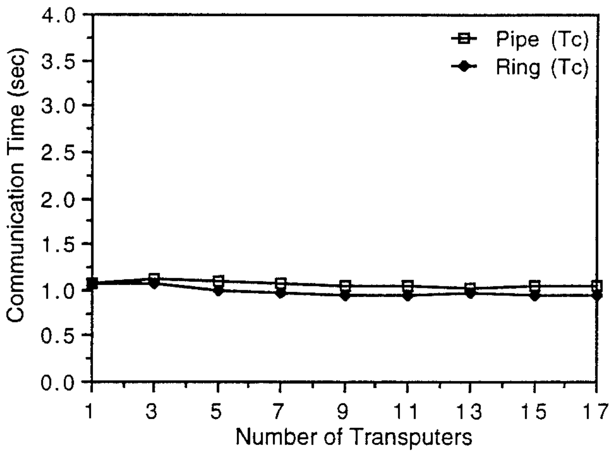
Table 6A: Eight Nearest Neighbor Mean Algorithm (PIPE) Results

#T800s	Tp(sec)	Tc(sec)	Tt(sec)	Computational Speed Up	Total Speed Up	Efficiency
1	2.482	1.064	3.546	1.00	1.00	1.00
3	0.781	1.114	1.896	3.18	1.87	0.62
5	0.469	1.096	1.565	5.29	2.27	0.45
7	0.336	1.060	1.396	7.39	2.54	0.36
9	0.263	1.054	1.317	9.44	2.69	0.30
11	0.218	1.049	1.267	11.38	2.80	0.25
13	0.182	1.022	1.204	13.64	2.95	0.23
15	0.155	1.049	1.204	16.01	2.95	0.20
17	0.136	1.052	1.188	18.25	2.98	0.17

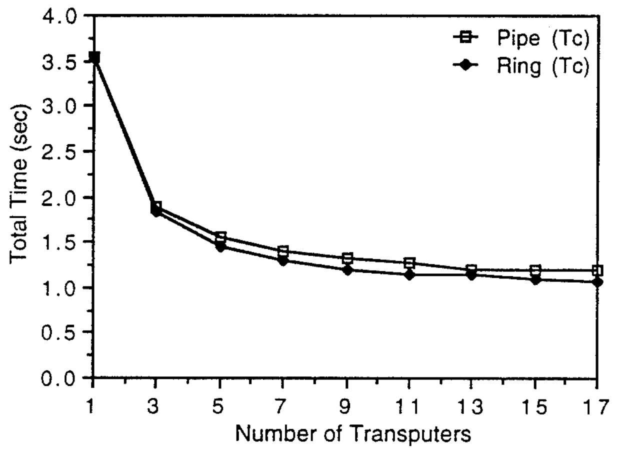
Table 6B: Eight Nearest Neighbor Mean Algorithm (RING) Results

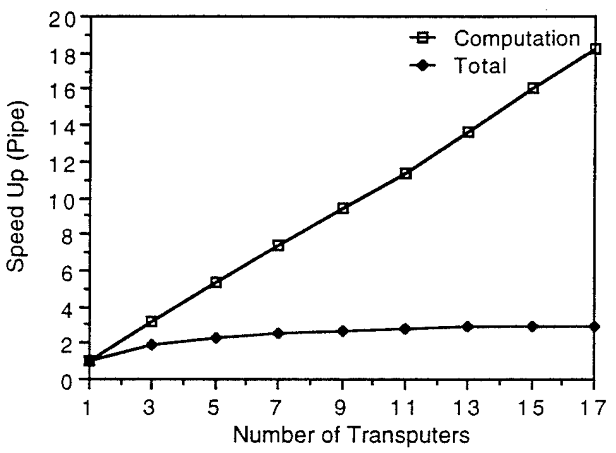
#T800s	Tp(sec)	Tc(sec)	Tt(sec)	Computational Speed Up	Total Speed Up	Efficiency
1	2.482	1.064	3.546	1.00	1.00	1.00
3	0.786	1.030	1.816	1.27	1.95	0.65
5	0.472	0.985	1.457	5.26	2.43	0.49
7	0.337	0.957	1.295	7.36	2.74	0.39
9	0.265	0.940	1.205	9.37	2.94	0.33
11	0.219	0.939	1.158	11.33	3.06	0.28
13	0.183	0.976	1.159	13.56	3.06	0.24
15	0.155	0.943	1.098	16.01	3.23	0.22
17	0.137	0.933	1.070	18.12	3.31	0.19



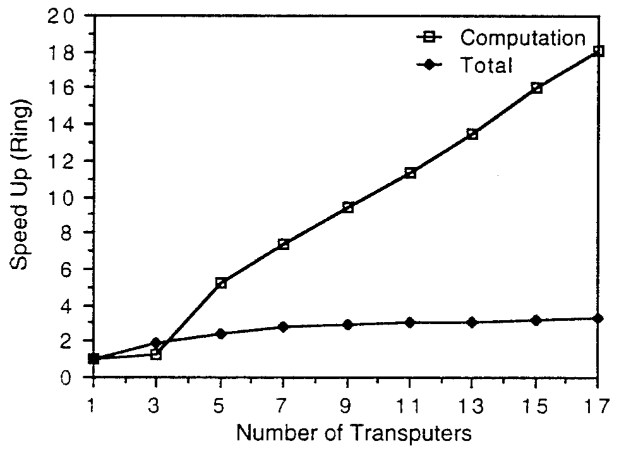


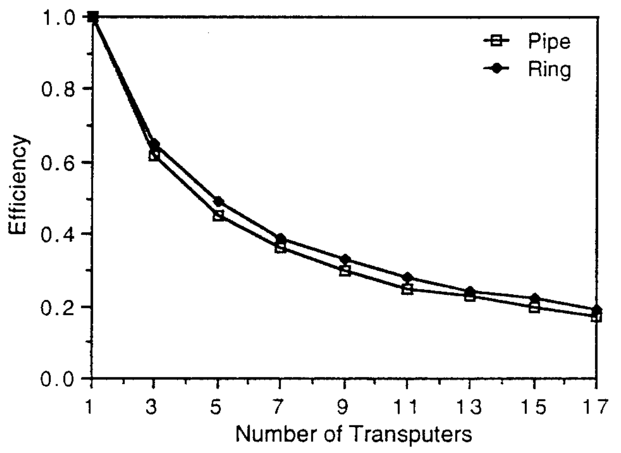
Figures 22A-22B: Figure 22A: Processing time curves for the pipe and ring architectures for the eight nearest neighbor mean algorithm. Figure 22B: Communication time curves for the same algorithm for both architectures.





Figures 22C-22D: Figure 22C: Total time curves for the pipe and ring architectures. Figure 22D: Speed Up curves for the algorithm using the pipe architecture.





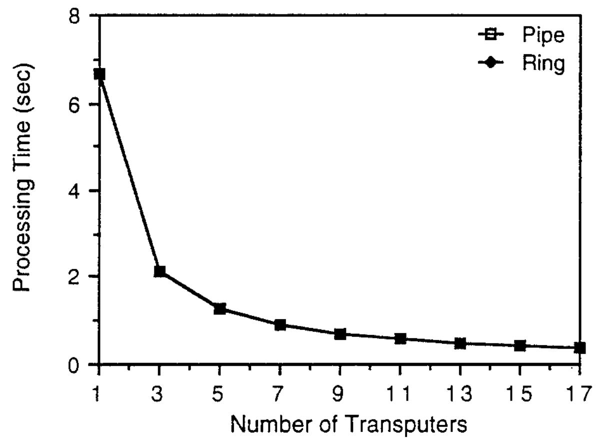
Figures 22E-22F: Figure 22E: Speed Up curves for the algorithm using the ring architecture. Figure 22F: Efficiency of the pipe and ring architectures using the total algorithm time.

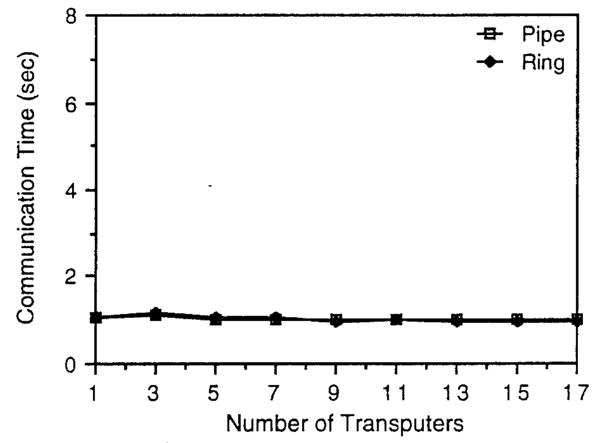
Table 7A: Four Nearest Neighbor Median Filter Algorithm (PIPE) Results

#T800s	Tp(sec)	Tc(sec)	Tt(sec)	Computational Speed Up	Total Speed Up	Efficiency
1	6.661	1.057	7.717	1.00	1.00	1.00
3	2.143	1.130	3.272	3.61	2.04	0.68
5	1.293	1.103	2.339	5.98	2.85	0.57
7	0.917	0.998	1.915	8.43	3.48	0.50
9	0.705	0.995	1.700	10.97	3.92	0.44
11	0.583	1.032	1.615	13.27	4.12	0.38
13	0.489	1.001	1.490	15.82	4.47	0.34
15	0.418	1.006	1.424	18.50	4.68	0.31
17	0.369	1.015	1.383	20.96	4.82	0.28

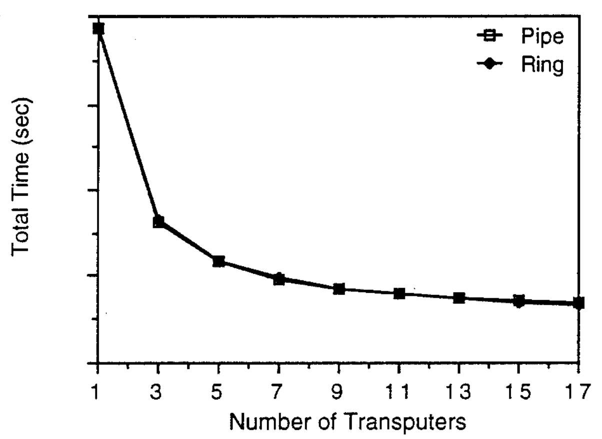
Table 7B: Four Nearest Neighbor Median Filter Algorithm (RING) Results

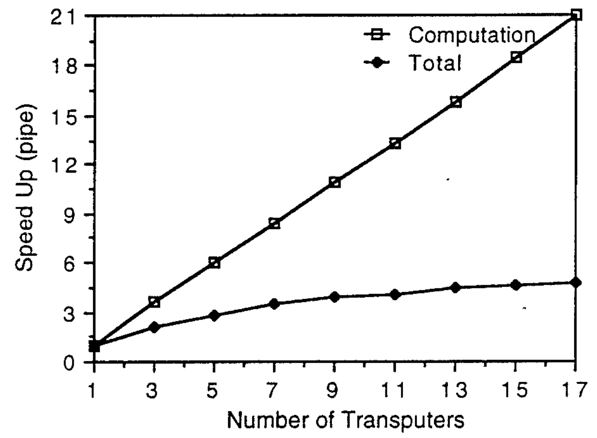
#T800s	Tp(sec)	Tc(sec)	Tt(sec)	Computational Speed Up	Total Speed Up	Efficiency
1	6.661	1.057	7.717	1.00	1.00	1.00
3	2.157	1.168	3.324	3.09	2.32	0.77
5	1.299	1.049	2.348	5.13	3.29	0.66
7	0.922	1.041	1.962	7.22	3.93	0.56
9	0.709	0.973	1.682	9.39	4.59	0.51
11	0.586	0.991	1.577	11.37	4.89	0.44
13	0.491	0.985	1.477	13.57	5.22	0.40
15	0.420	0.968	1.388	15.86	5.56	0.37
17	0.371	0.976	1.347	17.96	5.73	0.34



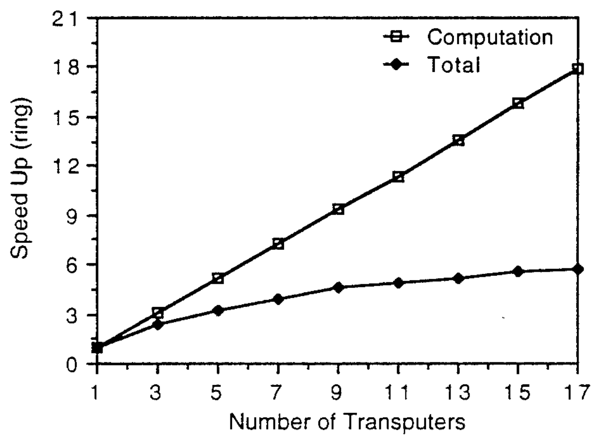


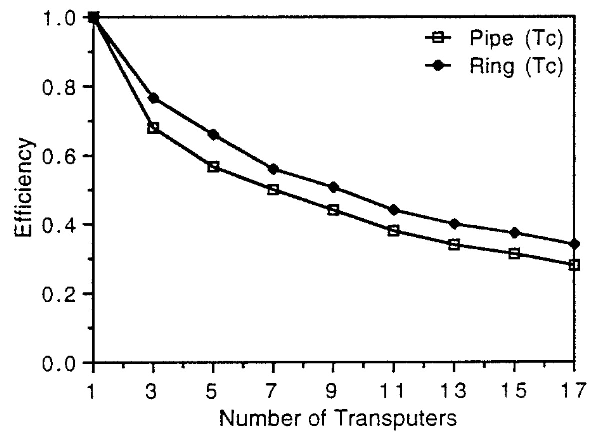
Figures 23A-23B: Figure 23A: Processing time curves for the pipe and ring architectures for the four nearest median algorithm. Figure 23B: Communication time curves for the same algorithm for both architectures.





Figures 23C-23D: Figure 23C: Total time curves for the pipe and ring architectures. Figure 23D: Speed Up curves for the algorithm using the pipe architecture.





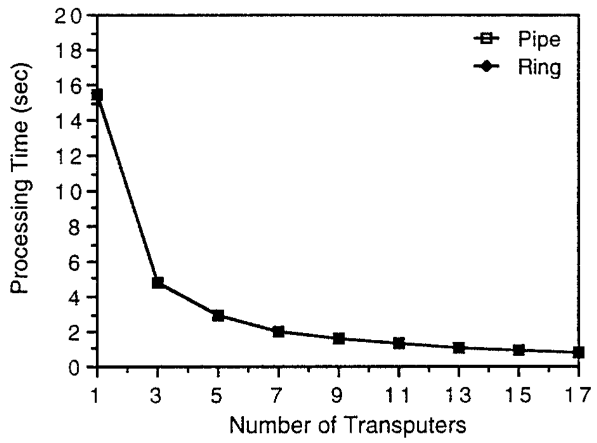
Figures 23E-23F: Figure 23E: Speed Up curves for the algorithm using the ring architecture. Figure 23F: Efficiency of the pipe and ring architectures using the total algorithm time.

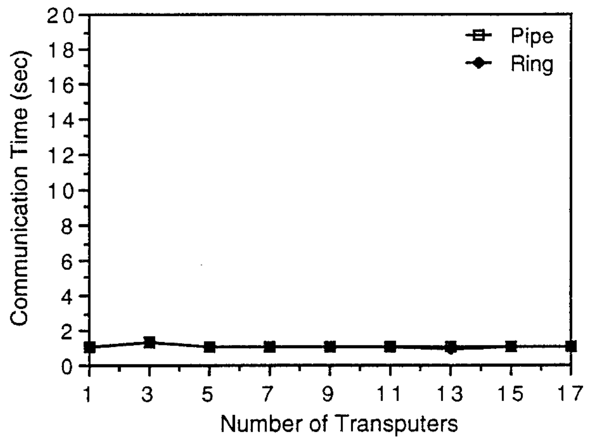
Table 8A: Eight Nearest Neighbor Median Filter Algorithm (PIPE) Results

#T800s	Tp(sec)	Tc(sec)	Tt(sec)	Computational Speed Up	Total Speed Up	Efficiency
1	15.499	1.102	16.551	1.00	1.00	1.00
3	4.796	1.330	6.127	3.23	2.70	0.90
5	2.900	1.112	4.013	5.34	4.12	0.82
7	2.049	1.074	3.123	7.56	5.30	0.76
9	1.575	1.077	2.652	9.84	6.24	0.69
11	1.306	1.042	2.348	11.87	7.05	0.64
13	1.095	1.011	2.106	14.15	7.86	0.60
15	0.939	1.045	1.988	16.50	8.33	0.56
17	0.282	1.054	1.878	18.72	8.81	0.52

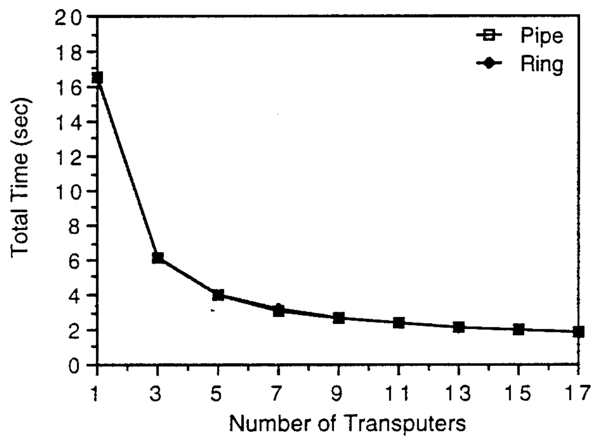
Table 8B: Eight Nearest Neighbor Median Filter Algorithm (RING) Results

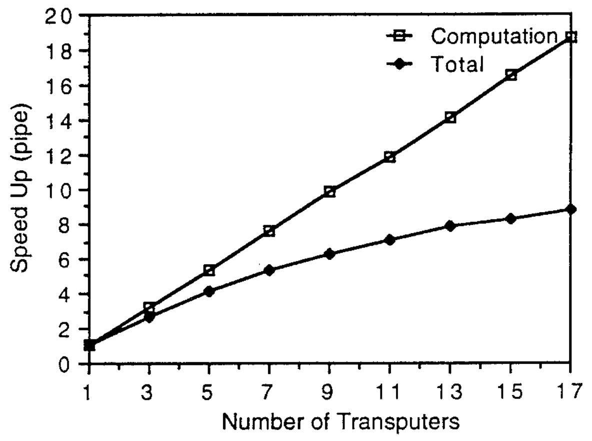
#T800s	Tp(sec)	Tc(sec)	Tt(sec)	Computational Speed Up	Total Speed Up	Efficiency
1	15.499	1.102	16.551	1.00	1.00	1.00
3	4.825	1.310	6.135	3.21	2.70	0.90
5	2.909	1.112	4.021	5.33	4.12	0.82
7	2.055	1.087	3.142	7.54	5.27	0.75
9	1.580	1.102	2.682	9.81	6.17	0.69
11	1.310	1.035	2.345	11.83	7.06	0.64
13	1.099	0.996	2.095	14.10	7.90	0.60
15	0.942	1.033	1.975	16.45	8.38	0.56
17	0.830	1.026	1.857	18.67	8.91	0.52



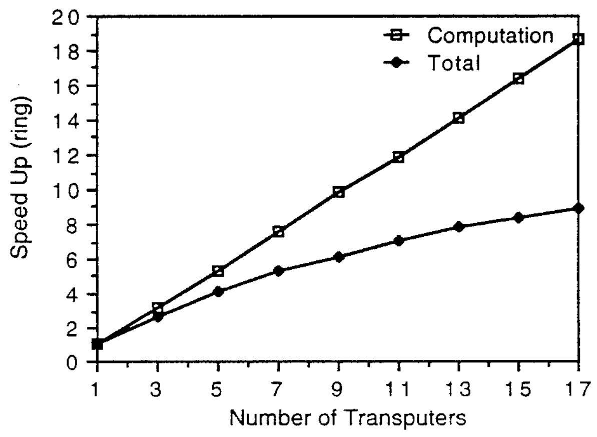


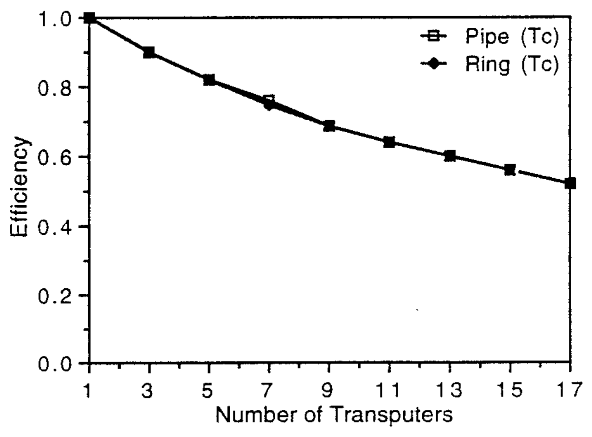
Figures 24A-24B: Figure 24A: Processing time curves for the pipe and ring architectures for the threshold algorithm. Figure 24B: Communication time curves for the same algorithm for both architectures.





Figures 24C-24D: Figure 24C: Total time curves for the pipe and ring architectures. Figure 24D: Speed Up curves for the algorithm using the pipe architecture.





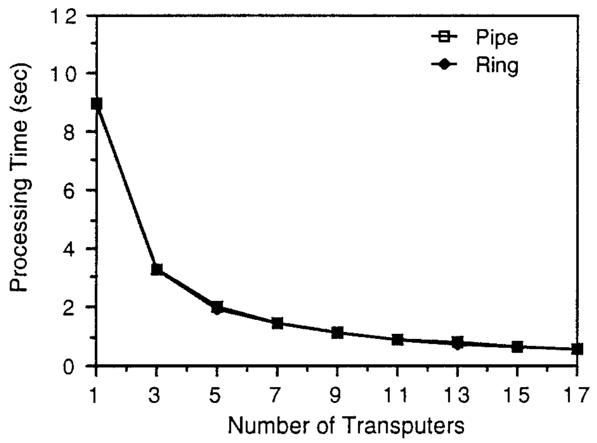
Figures 24E-24F: Figure 24E: Speed Up curves for the algorithm using the ring architecture. Figure 24F: Efficiency of the pipe and ring architectures using the total algorithm time.

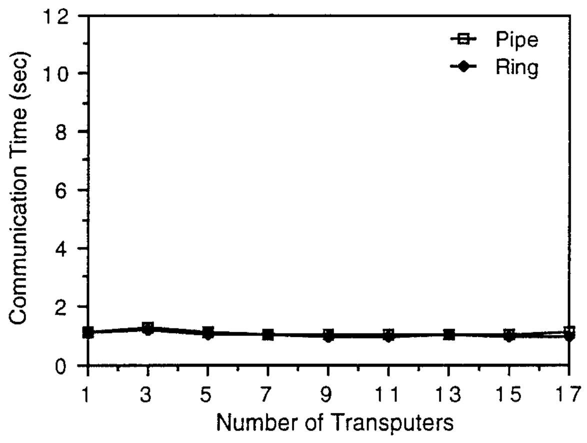
Table 9A: Sobel Filter Algorithm (PIPE) Results

#T800s	Tp(sec)	Tc(sec)	Tt(sec)	Computational Speed Up	Total Speed Up	Efficiency
1	8.977	1.085	10.062	1.00	1.00	1.00
3	3.272	1.250	4.522	2.74	2.23	0.74
5	1.963	1.123	3.086	4.57	3.26	0.65
7	1.411	1.604	2.475	6.36	4.07	0.58
9	1.106	1.052	2.158	8.12	4.66	0.52
11	0.915	1.062	1.977	9.81	5.09	0.46
13	0.763	1.039	1.802	11.77	5.58	0.43
15	0.648	1.060	1.708	13.85	5.89	0.39
17	0.572	1.085	1.657	15.69	6.07	0.36

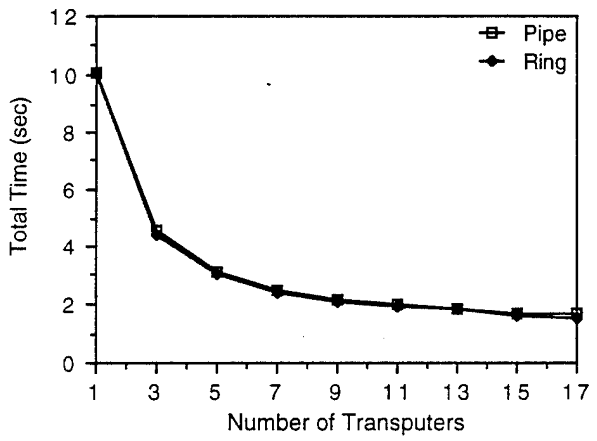
Table 9B: Sobel Filter Algorithm (RING) Results

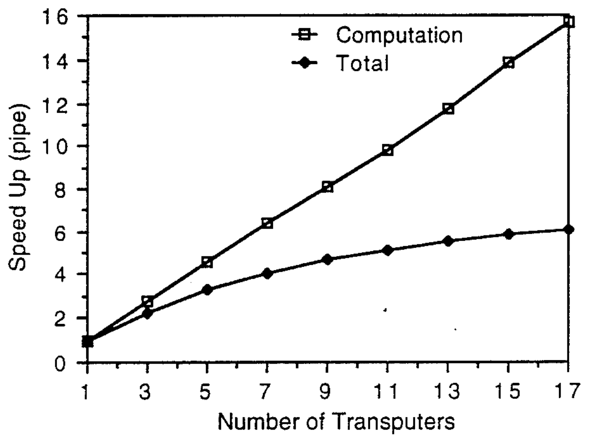
#T800s	Tp(sec)	Tc(sec)	Tt(sec)	Computational Speed Up	Total Speed Up	Efficiency
1	8.977	1.085	10.062	1.00	1.00	1.00
3	3.259	1.165	4.424	2.75	2.27	0.75
5	1.955	1.068	3.023	4.59	3.33	0.66
7	1.406	1.005	2.411	6.38	4.17	0.60
9	1.102	0.972	2.072	8.15	4.85	0.54
11	0.912	0.999	1.911	9.84	5.21	0.47
13	0.760	1.066	1.826	11.81	5.84	0.45
15	0.646	0.975	1.621	13.90	6.21	0.42
17	0.570	0.969	1.539	15.75	6.54	0.38



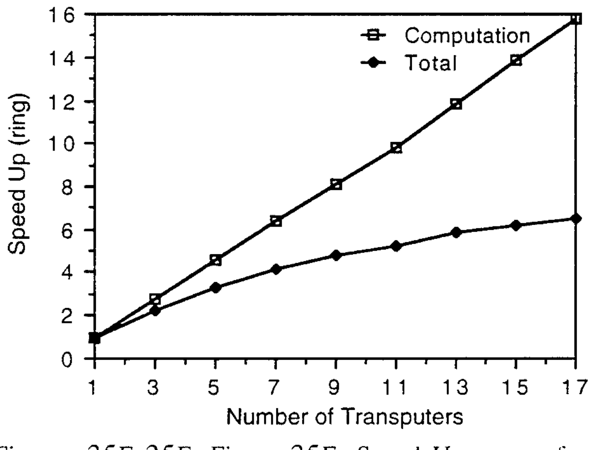


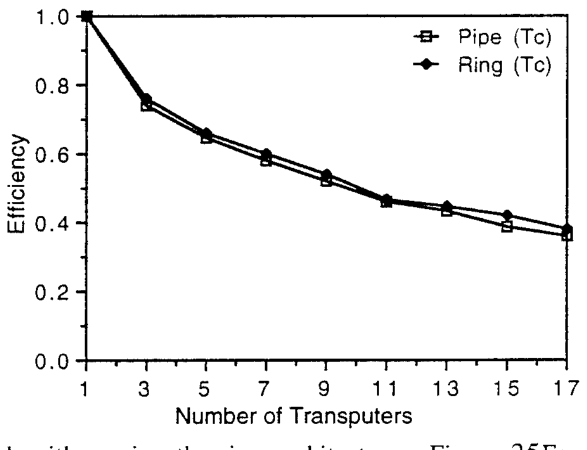
Figures 25A-25B: Figure 25A: Processing time curves for the pipe and ring architectures for the threshold algorithm. Figure 25B: Communication time curves for the same algorithm for both architectures.





Figures 25C-25D: Figure 25C: Total time curves for the pipe and ring architectures. Figure 25D: Speed Up curves for the algorithm using the pipe architecture.





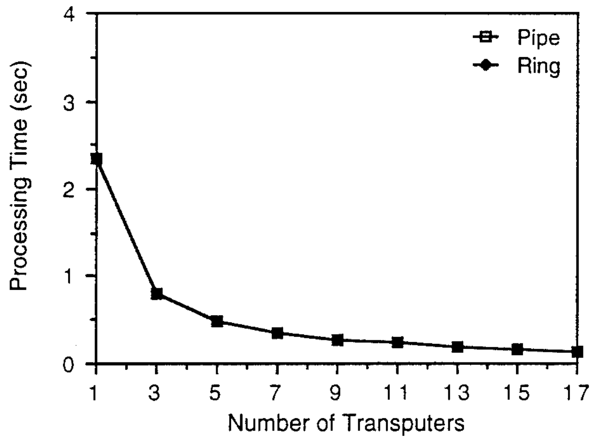
Figures 25E-25F: Figure 25E: Speed Up curves for the algorithm using the ring architecture. Figure 25F: Efficiency of the pipe and ring architectures using the total algorithm time.

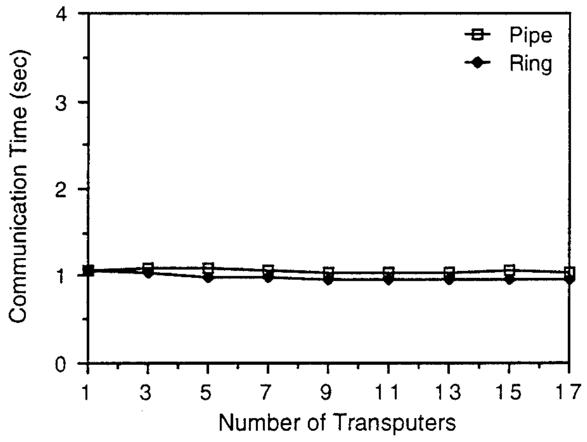
Table 10A: Roberts Cross Filter Algorithm (PIPE) Results

#T800s	Tp(sec)	Tc(sec)	Tt(sec)	Computational Speed Up	Total Speed Up	Efficiency
1	2.344	1.057	3.401	1.00	1.00	1.00
3	0.812	1.103	1.937	2.89	1.76	0.58
5	0.488	1.093	1.581	4.80	2.15	0.43
7	0.349	1.063	1.412	6.72	2.41	0.34
9	0.274	1.036	1.310	8.55	2.60	0.29
11	0.227	1.042	1.269	10.33	2.68	0.24
13	0.189	1.047	1.236	12.40	2.75	0.21
15	0.161	1.062	1.223	14.56	2.78	0.19
17	0.142	1.048	1.190	16.51	2.86	0.17

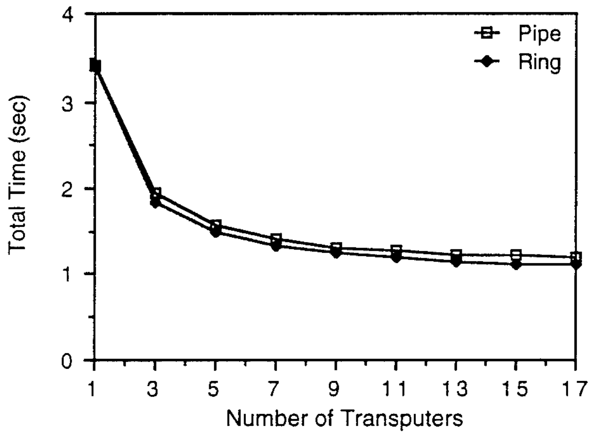
Table 10B: Roberts Cross Filter Algorithm (RING) Results

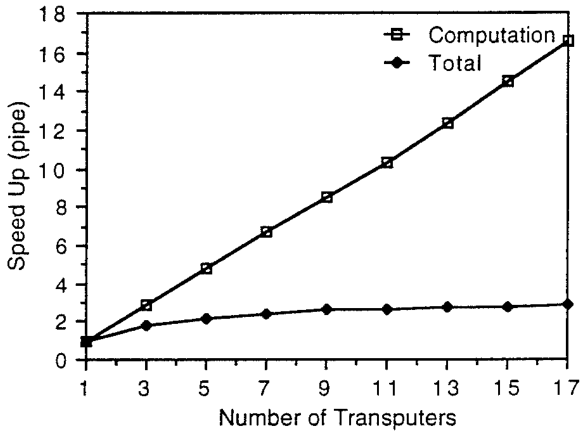
#T800s	Tp(sec)	Tc(sec)	Tt(sec)	Computational Speed Up	Total Speed Up	Efficiency
1	2.344	1.057	3.401	1.00	1.00	1.00
3	0.813	1.035	1.847	2.88	1.84	0.61
5	0.488	0.991	1.485	4.80	2.29	0.46
7	0.349	0.976	1.325	6.72	2.57	0.37
9	0.274	0.971	1.245	8.55	2.73	0.30
11	0.227	0.972	1.199	10.33	2.83	0.26
13	0.189	0.951	1.140	12.40	2.98	0.23
15	0.161	0.958	1.119	15.56	3.04	0.20
17	0.142	0.971	1.114	16.55	3.05	0.18



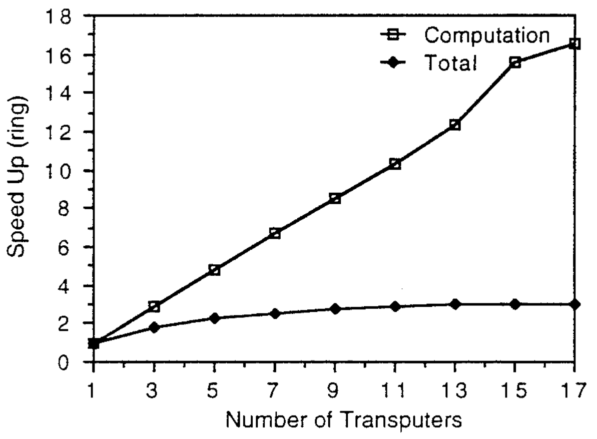


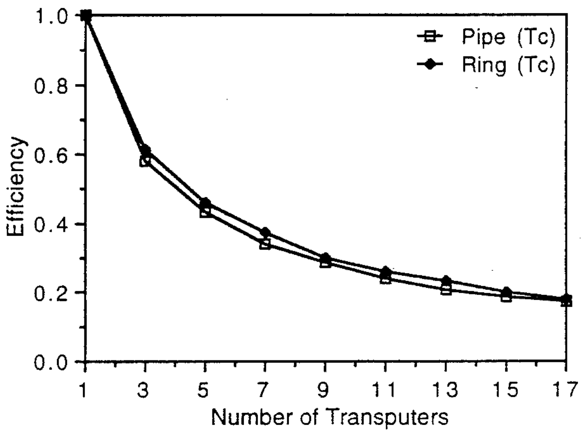
Figures 26A-26B: Figure 26A: Processing time curves for the pipe and ring architectures for the threshold algorithm. Figure 26B: Communication time curves for the same algorithm for both architectures.





Figures 26C-26D: Figure 26C: Total time curves for the pipe and ring architectures. Figure 26D: Speed Up curves for the algorithm using the pipe architecture.





Figures 26E-26F: Figure 26E: Speed Up curves for the algorithm using the ring architecture. Figure 26F: Efficiency of the pipe and ring architectures using the total algorithm time.

8.0 Analysis

The analysis is broken up into two different sections. The first section analyzes the problems encountered in the project in general, and the second portion analyzes the results obtained from the parallel implementation.

8.1 Problems Encountered

Five basic problems arose when mapping the serial algorithms from the single transputer algorithms into parallel algorithms on multiple transputers. Here is a detailed description of the problems encountered.

8.1.1 Debugging

Although Occam comes with an excellent debugging tool, there is more to it than that. Since only the root transputer is connected to the host, this is the only transputer that can perform I/O. The other transputers can not write to the screen. This has presented a problem when attempting to debug a program utilizing the seventeen transputers. Since variables cannot be written to the screen to see what their values are to assist in debugging, STOP (break) statements must be inserted in the code to check the variables. This has proven to be a time consuming chore. So even the tiniest error, if not obvious, requires painstaking debugging techniques.

8.1.2 Global Normalization

Some routines require global normalization, since after computation they have very large negative values and very large positive values. Each transputer calculates its own local minimum and maximum variables, and each of these values is different for each transputer. If each transputer normalized its portion of the image it has stored in memory using its local values, the resulting image would be chunks of columns with varying shades of grey. The local minimum and maximum values from each transputer must first be calculated then passed through the network back to the root transputer through the channels connecting them, comparing them along the way to obtain the global values. Once the global values are obtained, these values must be passed back to all of the transputers, so they can all normalize with the same minimum and maximum values.

8.1.3 Lines In The Output Image (Partitioning the Data)

A problem arose where vertical black lines were appearing in the output image. This was due to the fact that the output image was being initialized to be zero all over to avoid erroneous values at the borders that are not calculated. Then when the calculations were performed, not enough columns were being processed on each transputer, therefore resulting in columns remaining at value zero (similar to figure 11). This problem was corrected and all routines are now producing smooth images with no lines in them.

8.1.4 Deadlock

Deadlock occurs when two or more concurrent processes are stuck waiting for each other due to a communication interdependency. The most common deadlock occurring is when a transputer sends data over a channel, and another transputer is waiting to receive data on a different channel, thus, the two are stuck waiting. All channels must be declared, and data types must match when passing data among processors.

Each transputer has at least one pair of unidirectional connections from its own link to the connecting transputer's link. The connecting pairs make up a "channel" of communication between transputers. Data can only flow in one direction on these links, and on most of the transputers, there are two channels connecting more than one transputer. Data flows "out" of one transputer and "in" to another in appropriate links. Once the links have been established for data flow, the data cannot flow against the original configuration. Hence, they are unidirectional links that always flow in the same direction. A problem encountered in this project was that data was being sent out of one transputer, and waiting to be received at another transputer, and hanging in the process. In debugging, it hung right at the send command in one transputer, and right on the receive command at another transputer. So the program was stuck sending data, and receiving it at the same time, which did not make sense. The problem was that data was being sent out of one channel, and waiting to be received from another channel on another link of the transputer, so the data essentially would never get there. Once the problem was discovered, it was quite simple to listen from the correct channel to receive it and solve the problem. It was sometimes essential in debugging to physically draw out the network and its channels, to visualize the path of data flow.

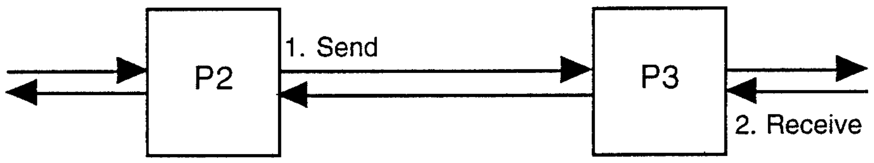


Figure 27: Deadlock between two transputers. Processor P2 sends data to processor P3, which attempts to receive the data along a different channel than the data was sent on. Both processors are in a state of deadlock and cannot proceed.

8.1.5 Type Casting

Occam is a very type specific language. Every variable has to be declared as a specific type, and different types cannot be used in conjunction with each other. Occam does allow type conversion, where a value of a primitive type can be converted to a numerically similar value of another primitive type. The image files I have that need to be displayed are data type BYTE, which ranges from 0 to 255. In order to do any type of arithmetic operations, the image values must be converted to data type INT, and to perform more intricate operations, the data must sometimes be converted to REAL. All these conversions get confusing, and the complier makes sure they're all properly done with no exceptions. Although these conversions save having to declare several extra variables, it becomes time consuming and sometimes confusing because of the strict compiler. The whole point of types is to prevent values being used in inappropriate situations, so I guess this type of problem is common.

8.2 Parallel Implementation Analysis

This section will analyze the actual results achieved, by analyzing the graphs plotted in section seven. There was no significant difference between the straight pipe network and the ring network, due to the vast amount of data shuffling that was required.

8.2.1 Speed-Up

Initially, it was predicted that these selected algorithms would exhibit a near-linear speed-up when executed in parallel on multiple transputers. By analyzing the graphs from the previous section, it is clearly not the case. Most algorithms exhibited a near-linear speed-up in the computations, but for the total time speed-up quickly peaks and levels off.

8.2.1.1 Factors That Limit Speed-Up

A parallel algorithm can exhibit constrained speed-up if there isn't enough work to be done by the number of available processors, thus creating idle processors, which is more commonly referred to as Amdahl effect. The size of the input problem could also have a serious effect on limiting the speed-up of an algorithm. In this particular case, the communication times of the algorithms is clearly the dominating factor in the total time of the algorithms. Since communication cannot be ignored when calculating the speed-up, the near-linear speed-up achieved by the array of transputers is irrelevant.

8.2.2 Ease-Of-Implementation

The Occam language was difficult to grasp at first. Though once the core algorithms were set up for properly partitioning the data, the rest of the job was quite easy. The debugger was a big help, and usually was thorough enough to solve most problems encountered. The biggest stumbling block of this project was first to figure out how the data was to be partitioned, then to get it to its designated transputer as efficiently as possible. Although the entire image could have been sent to each transputer, it would have been much more inefficient to do so. Instead, only the required chunks of imagery were sent.

8.2.3 Efficiency

As we can see by the efficiency graphs from the previous section, efficiency was drastically reduced as the number of processors increased. Although the rules for effective parallel processing were closely followed by minimizing the number of process creations and synchronizations, communication overhead was extremely large due to the size of the data in an image. The amount of work done between synchronizations was as large as possible, while keeping all the processors busy, but more time was spent passing and receiving the image through the network than was done in actual processing.

8.2.4 Suggested Improvements

Since the cost of communication is so high when dealing with imagery, the only suggested improvement to achieve greater speed-up and increased efficiency is to perform more processing on an image while it is in the memory of the transputer. As we can see by the speed-up graphs and efficiency graphs, the algorithms that had to do more calculations achieved a greater speed-up per processor as well as increased efficiency. For example, the eight nearest neighbor median algorithm had to perform the most calculations, and achieves the greatest results (see Figures 24A-F). Each pixel processed in this algorithm had to be sorted with its eight nearest neighbors to obtain the new value.

This selection of algorithms used in this project is good representation of commonly used algorithms in the image processing arena, they are more commonly used together. For example, an initial image may be smoothed by an averaging routine, then applied some form of edge detection, then thresholded at a particular value. This would greatly increase the speed-up and efficiency of the routine by performing various operations one after another while the image was in the transputers' memory.

9.0 Conclusions

Since this report was started, the transputers are close to being obsolete. Computer technology has advanced tremendously since the initial purchase of these transputers, and prices have been reduced drastically. Single processor chips capable of handling over 2 billion operations per second are now commercially available at an affordable price, thus making the need for a large number of processors unnecessary. In some instances there is no longer a need for more than a single processor with this level of power.

Although this technology is old at the point of concluding this experiment - this experiment was far from a waste of time. The project has been helpful in understanding the variety of problems associated with image processing that are taken for granted by the user, such as the highly complicated process of breaking the image up, calculating the required overlap, processing it, and reassembling it correctly. These insights give a better appreciation of what goes on "behind the scenes" in parallel algorithms. It also helps the programmer realize why writing efficient programs is so important and how useful a debugging tool is.

10.0 Bibliography

1. SGS-Thompson, The Transputer Databook, Second Edition, 1989, Inmos Limited.
2. Quinn, M.J., *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill Inc., 1987.
3. Inmos, Occam 2 Reference Manual, Inmos Limited, Prentice Hall International, 1988.
4. Pountain & May, A Tutorial Introduction to Occam Programming, Inmos Limited, BSP Professional Books, 1987.
5. Babb, Robert G., II., *Programming Parallel Processors* Addison-Wesley Publishing Co., 1988.
6. Hwang, K.; Briggs, Faye A., *Computer Architecture and Parallel Processing* McGraw-Hill Inc., 1984.
7. Tiele, H.J.J. Te; Dekker, Th.J.; Van Der Vorst, H.A., *Special Topics in Supercomputing*, Volume 3, Algorithms and Applications on Vector and Parallel Computers, Elsevier Science Publishers B.V., 1987.