(2)

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

SENSITIVITY ANALYSIS OF
TRANSPUTER WORKFARM TOPOLOGIES

by

Timothy J. Johnson

September 1989

Thesis Advisor: Chyan Yang

Approved for public release; distribution unlimited

Unclassified

Security Classification of this page

REPORT DOCUMENTATION PAGE

1a Report Security Classification: Unclassified			1b Restrictive Markings
2a Security Classification Authority			3 Distribution Availability of Report: Approved for public release; distribution is unlimited.
2b Declassification/Downgrading Schedule			
4 Performing Organization Report Number(s)			5 Monitoring Organization Report Number(s)
6a Name of Performing Organization: Naval Postgraduate School	6b Office Symbol (*If Applicable*): 62		7a Name of Monitoring Organization: Naval Postgraduate School
6c Address (*city, state, and ZIP code*): Monterey, CA 93943-5000			7b Address (*city, state, and ZIP code*): Monterey, CA 93943-5000
8a Name of Funding/Sponsoring Organization	8b Office Symbol (*If Applicable*)		9 Procurement Instrument Identification Number
8c Address (*city, state, and ZIP code*)			10 Source of Funding Numbers: Program Element Number / Project No / Task No / Work Unit Accession No
11 Title (*Include Security Classification*): Sensitivity Analysis of Transputer Workfarm Topologies			
12 Personal Author(s): Timothy J. Johnson			
13a Type of Report: Master's Thesis	13b Time Covered: From To	14 Date of Report (*year, month,day*): September 1989	15 Page Count: 79
16 Supplementary Notation: The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
17 Cosati Codes: Field / Group / Subgroup		18 Subject Terms (*continue on reverse if necessary and identify by block number*): Network, Workfarm, Load Balancing, Linear Network, Tree Network, Transputers, Multiprocessors	

19 Abstract (*continue on reverse if necessary and identify by block number*)

Parallel processing structures such as multiprocessor arrays and pipelining enhance throughput tremendously for suitable algorithms having high degrees of concurrency. However, if the time to process different workpackets becomes irregular, much of the advantage offer traditional sequential processing systems may be lost.

In an attempt to produce a more flexible response to workload demands, a transputer workfarm was investigated. Two network topologies, a linear model and a tree model, were built using the transputer as the processing element (PE), or worker. An algorithm was developed which could be run independently on all workers in the workfarm. Each worker produced results independent of the other workers. By altering specific variables within the algorithm, the network performance could be changed. The results from this thesis illustrate how these parameters affect each network and provide comparative information between the linear model and the tree model.

20 Distribution/Availability of Abstract: [X] unclassified/unlimited [ ] same as report [ ] DTIC users	21 Abstract Security Classification: Unclassified	
22a Name of Responsible Individual: Chyan Yang	22b Telephone (*Include Area code*): (408) 646-2266	22c Office Symbol: 62YA

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted

All other editions are obsolete

security classification of this page: Unclassified

Approved for public release; distribution is unlimited.

Sensitivity Analysis of

Transputer Workfarm Topologies

by

Timothy J. Johnson
Lieutenant, United States Navy
B.S., Norwich University, 1980

Submitted in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 1989

Author: _____
Timothy J. Johnson

Approved by: _____
Chyan Yang, Thesis Advisor

_____
Man-Tak Shing, Second Reader

_____
John P. Powers, Chairman, Electrical and Computer Engineering

ABSTRACT

Parallel processing structures such as multiprocessor arrays and pipelining enhance throughput tremendously for suitable algorithms having high degrees of concurrency. However, if the time to process different workpackets becomes irregular, much of the advantage offer traditional sequential processing systems may be lost.

In an attempt to produce a more flexible response to workload demands, a transputer workfarm was investigated. Two network topologies, a linear model and a tree model, were built using the transputer as the processing element (PE), or worker. An algorithm was developed which could be run independently on all workers in the workfarm. Each worker produced results independent of the other workers. By altering specific variables within the algorithm, the network performance could be changed. The results from this thesis illustrate how these parameters affect each network and provide comparative information between the linear model and the tree model.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

I would like to thank several people who contributed to the completion of this thesis. First, my advisor Prof. Yang. He provided me with a great deal of guidance including new ways to look at the problem. Without his help I would have had much greater difficulty in completing as much as I was able to. Secondly I would like to thank Captain Rod Scott, RCAF, for providing a source code base for part of the linear model from which to work from. Next, Prof. Kodres' generosity in providing the facilities, transputers, computers, and study space without which none of this would have been possible. Finally, to Prof. Shing for being able to fulfill the requirements of the second reader resulting from Prof. Kodres' untimely departure for surgery.

DEDICATION

To my wife Sandie who was my inspiration for completing my degree and kept me looking to the future. She was very supportive throughout my study program despite having to handle her own career and was especially strong during the last four months when we were separated because to her career requirements. She persevered through this and was still able to offer me encouragement.

I. INTRODUCTION

A. BACKGROUND

Distributed computing systems provide an exciting avenue for enhancing performance of hardware and software systems. Traditional multiprocessor systems utilize a one- or two-dimensional array of processors using nearest-neighbor or pipeline computing structures [Ref. 1]. Many networks will operate efficiently when each processor shares the computational load, i.e., achieves load-balancing. Programmers and users have a unique environment in which algorithms possessing a concurrent nature can be run with much higher efficiencies than previously encountered in sequential programming structures. Most solutions to existing problems have been approached from a sequential processing aspect. A significant percentage of these problems have varying degrees of underlying concurrency which may be exploited. The most dramatic advantage to be gained by exploiting the concurrent nature of the algorithms is the increased throughput. By dividing a task into several smaller tasks and processing them concurrently on separate processing elements (PEs), a tremendous decrease in processing time may be observed.

Once a task has been determined to have some inherent concurrency, either a system must be built to conform to the nature of the problem, or the problem must be configured to run on existing parallel processing systems. Stone, [Ref. 1], points out that there are several different parallel processing structures and philosophies

available, each with its own distinct advantage. The purpose of this thesis is to investigate one of the structures called a workfarm. The PE's within the workfarm may be configured into many topologies and therefore only two will be looked at in depth, the linear topology and the tree topology.

B. WORKFARM CONCEPT

Suppose a problem can be broken into a finite number of identical parts, each of which takes a different amount of time to solve. Due to varying processing times, nearest-neighbor or pipeline designs may encounter difficulties caused by an unbalanced work load on adjacent processors resulting in communications delays.

A logical alternative is a processor workfarm. In a workfarm, each processor (worker) executes the same functional block of code on individual workpackets independently of adjacent workers. This design is inherently different from the nearest-neighbor arrays and pipelines in that adjacent workers operate asynchronously with respect to each other. A controlling processor distributes the workpackets to the workfarm whenever a results packet is returned from the network. This synchronization is handled within the software at the controller level.

In the case of a linear model, Figure 1.1, a controller distributes workpackets to the first worker in the workfarm. The first worker will process as many packets as it is able to handle and send the remaining workpackets on to the rest of the workfarm. This happens

for each successive worker until the end worker is encountered. Results are returned to the controller by trickling back up the linear network in the opposite direction to the flow of work.

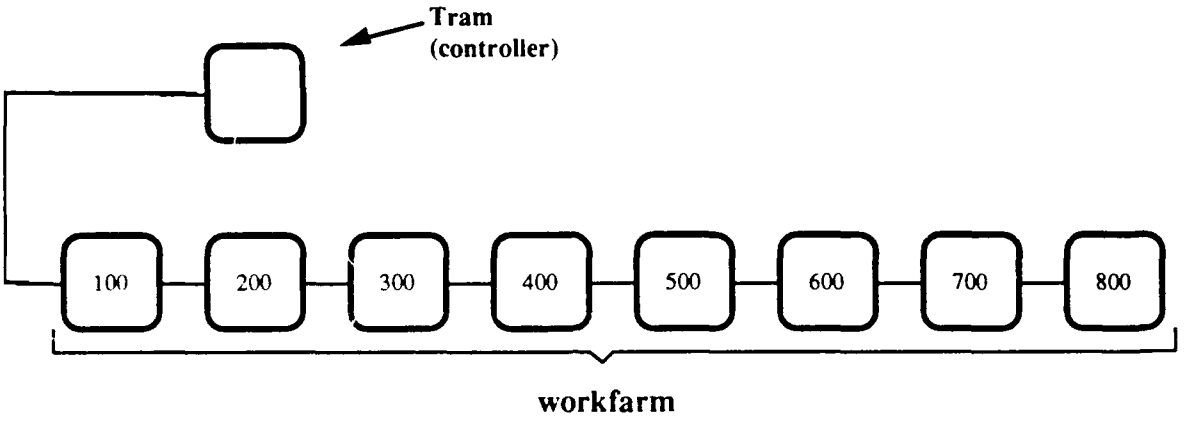


Figure 1.1 **Linear Model Workfarm**

The sensitivity of a given topology is analyzed by altering various parameters within the algorithm which might affect the performance of the workfarm. Parameters which may be altered to observe this effect include the input buffersize of each worker (how many workpackets a worker may buffer before having to pass additional workpackets on), the size of the workpackets (how many iterations must be done per workpacket), etc. The results presented in this thesis indicate that buffersize and the size of the workpackets (stripsize) are limiting factors in the linear model workfarm. In addition, two modes of operation for the linear model have been studied, non-addressed and addressed. In both cases the controller sends a new workpacket out to the network each time it receives results from a previously processed workpacket. In the non-

addressed mode the first available worker encountered by the workpacket will grab the workpacket for processing or buffering, if sufficient space is available. On the other hand, the same controller in the address mode will send the workpacket directly to the specific worker in the chain which just returned a results packet. Each of these designs may have its own advantages.

II. THE TRANSPUTER

A. OVERVIEW

The T800 [Ref. 2] transputer is just one in a family of transputers produced by INMOS. The parallel architecture, augmented with the CSP-based (Communicating Sequential Processes) [Ref. 3] language, OCCAM [Ref. 4], makes it an ideal and inexpensive tool to conduct research in topics of concurrency. Due to the serial link intercommunication structure of the transputer, a variety of network topologies can be easily implemented. Prior to discussing these topologies, a greater understanding of transputer architecture is required.

B. ARCHITECTURE

The T800 is a single chip implementation of what is traditionally a separate microprocessor and several support chips. Figure 2.1 shows a simple breakdown of the T800 architecture. Transputer regions are subdivided into a RISC technology microprocessor, on-chip RAM, four paired input/output serial links, an external memory interface module and a systems services module. In addition the T800 model incorporates an on-chip floating point unit.

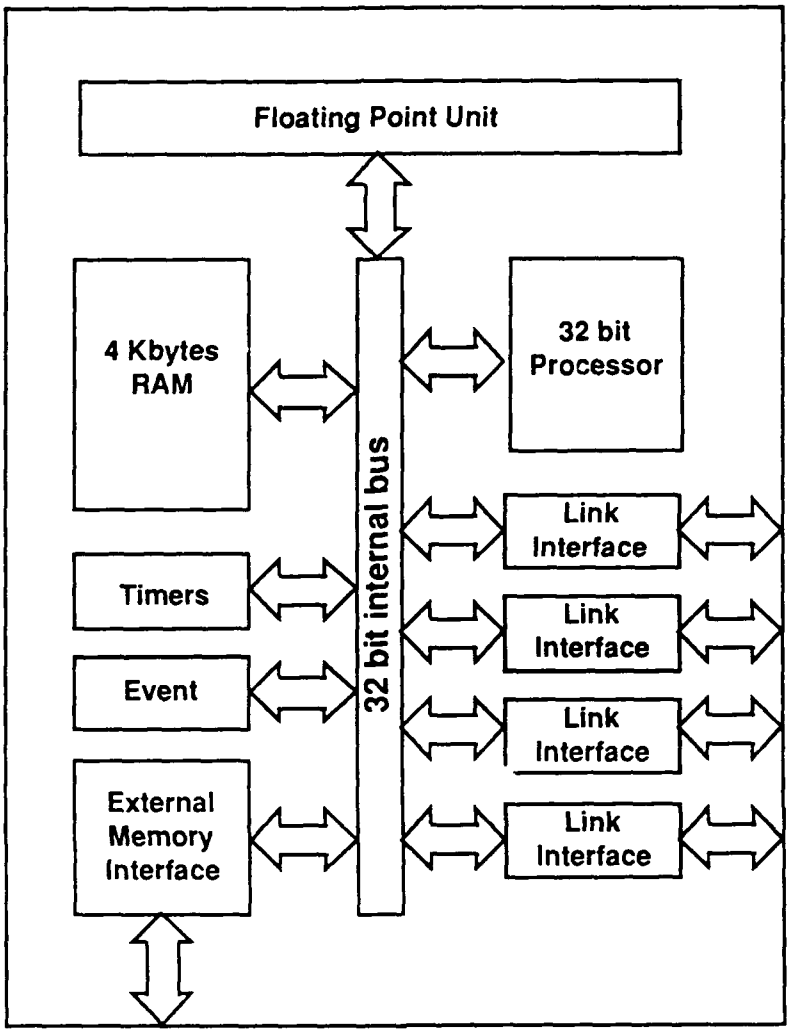


Figure 2.1 **T800 Transputer Block Diagram**

1. Processor

The processor is a RISC machine containing instruction logic, an instruction pointer, a workspace pointer, an operand register and three source/destination stack-like registers. All registers are 32 bits long. Four Gbytes of memory may be addressed. The first 4 Kbytes of

this address space are on-chip RAM. The various registers have the following functions:

- workspace pointer - points to an area of memory containing local variables
- instruction pointer - points to the next instruction to be executed
- operand register - used to form instruction operands
- A, B, C registers - evaluation stack

a. Instructions

Instructions refer to the stack implicitly. For example, evaluation of an add instruction adds the contents of A to the contents of B and places the sum in A. Overflow protection is not provided in hardware as this is easily handled by the compiler.

All instructions in the instruction set have the same format and are representatives of the most commonly used instructions in most programs. Each instruction is a single byte which can be decomposed into two 4 bit segments. The upper 4 bits contain the function code and the 4 lower order bits are a data value. This alone limits the number of functions to 16. However, most program operations involve the loading of small literal values and the loading and storing of one of a small number of variables. Two of the 16 functions, prefix and negative prefix, provide for extending the length of the instruction operand. The prefix instruction first loads its 4 data

bits into the operand register and then shifts this value to the left 4 bits. The negative prefix instruction merely complements the operand register prior to executing the shift. This scheme can create operands in the range of -256 to 255 by simply using just one of the appropriate prefix instructions.

b. Concurrency

The fundamental programming structure is known as a process, which is simply a sequence of instructions. A single transputer can run concurrent processes independent of a network of transputers. This is allowed by multiplexing high and low priority processes. Low priority processes are run whenever high priority processes are idle or are waiting for communications. Typically high priority processes are of short duration while most low priority processes are of longer durations. The user defines what processes run as high priority and which run as low priority. These are user-defined and eliminate the need for a kernel.

2. FPU

The T800 also houses a 64 bit floating point unit. The FPU performs single and double length arithmetic conformed to floating point standard ANSI-IEEE 754-1985. This FPU is capable of sustaining 2.25 MFLOPS processing concurrently with the CPU on a 30 MHz transputer model. However, for this thesis the T800 was operated at 20 MHz.

3. Memory

The T800 is configured with 4KBytes of on chip static random access memory. This memory serves as the lowest address block for the 4 GBytes of memory addressable by the T800. The remainder of the 4 GBytes must be supplied as external memory via a 32 bit bus. The 4KBytes of on chip RAM may be accessed via the 32 bit internal bus for read/write operations in one clock cycle.

4. Serial Links

The four pairs of input/output serial links provide the means for building networks of processors. These links allow the implementation of CSP by providing a means to make direct communications channels between processors within the network.

5. Timers

There are two hardware timers within the T800 which operate at two distinct levels. The high level timer provides 1 μsec ticks for the system whereas the low priority clock produces ticks of 64 μsec intervals.

III. WORKFARM TOPOLOGY

A. Overview

The workfarms to be investigated in this thesis required the ability to be easily configured into a variety of network topologies. The T800 transputer and its programming language, OCCAM, were chosen because of the simplicity in implementing multiprocessor networks with them. As previously stated, all parallel processing structures have advantages and disadvantages specific to their design. For this reason two different topologies have been studied in this thesis to determine their efficiencies relative to a given independent algorithm as the test-bed. The two topologies chosen were the linear model and the tree model. Since the goal was to determine the most efficient model for the given independent algorithm, each model was tested by varying specific parameters within the algorithm which might have influence on the interprocessor communications and actual time spent doing calculations. In each case the desired goal was to minimize the time to complete a given batch of work and to observe the resulting load balance for all workers in the network. The models are looked at in more detail in the following sections.

Given a sufficient number of transputers, imagination is the only limit to the number and design of possible topologies. However, practicality from a hardware and software implementation standpoint would argue that more regular and symmetrical structures be investigated. With regards to these considerations, two other

topologies were of interest given the number of transputers available for this network study. However, due to time considerations, a loop model and a cube model could not be implemented. A basic discussion of the loop and cube topologies will be included in the next section, but it will be limited to hardware configuration only.

B. Models

1. Linear

The linear model is a straight forward application of the transputers located on the B003 boards and the TRAM. A block diagram of the linear topology is shown in Figure 1.1. The model is implemented by simply connecting the transputers in a one dimensional array and ensuring that the appropriate link connections are defined in the software. A diagram of the linear workfarm model is shown in Figure 3.1.

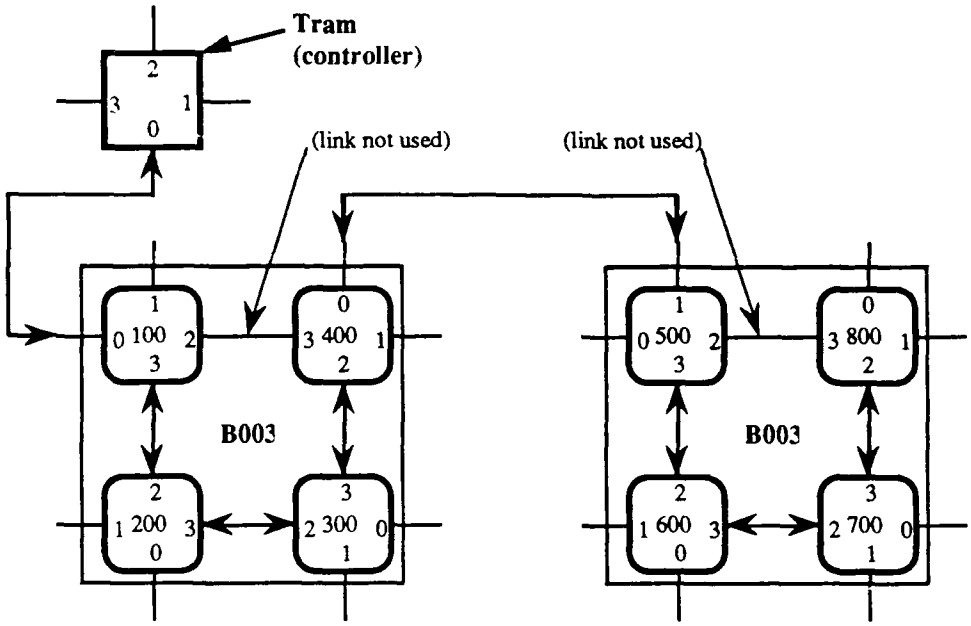


Figure 3.1 **Linear Model Physical Link Diagram**

The arrows depict the direction of data flow along the links. Note that these are not bidirectional links, but rather both the in and out complements for each link are used as a pair to provide output and input communications. Workpackets are generated in the controller and sent to the network via the outward flow of the links. Results are passed back up the links to the controller in the other direction.

2. Tree

The tree model is a more significant deviation from a simple linear design topology. The TRAM has three pairs of output and input links to choose from for connecting to the workfarm network. For the tree model, two of these link pairs are used. Each link pair connects to one of the B003 boards. The root worker of the B003 board already

has links to its two orthogonally adjacent neighbors and a simple hardwire connection is made from its fourth available link pair to the diagonally positioned worker on the board. This configuration creates two branches from the TRAM with each branch splitting into three more branches. The configuration is shown abstractly in Figure 3.2 and schematically in Figure 3.3.

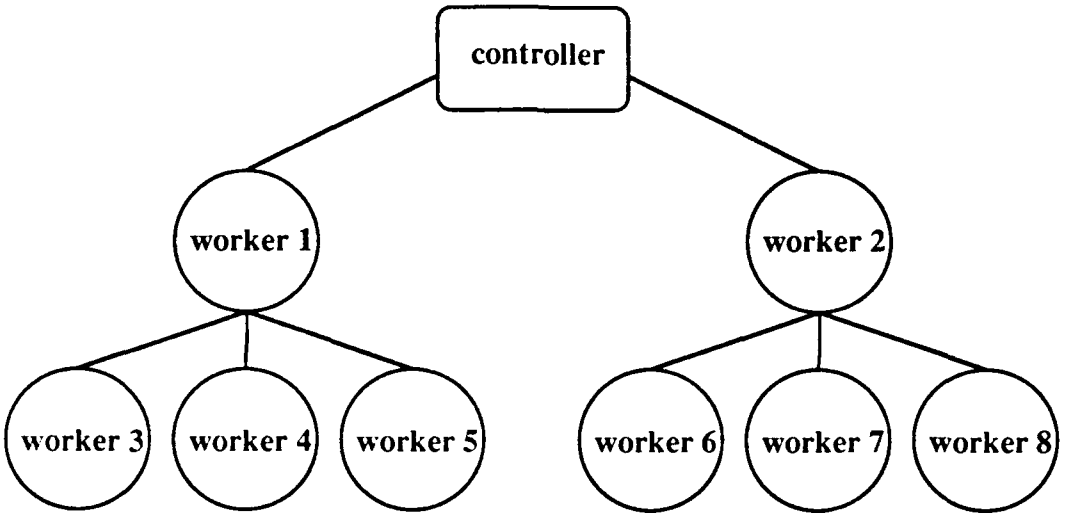


Figure 3.2 **Tree Model Block Diagram**

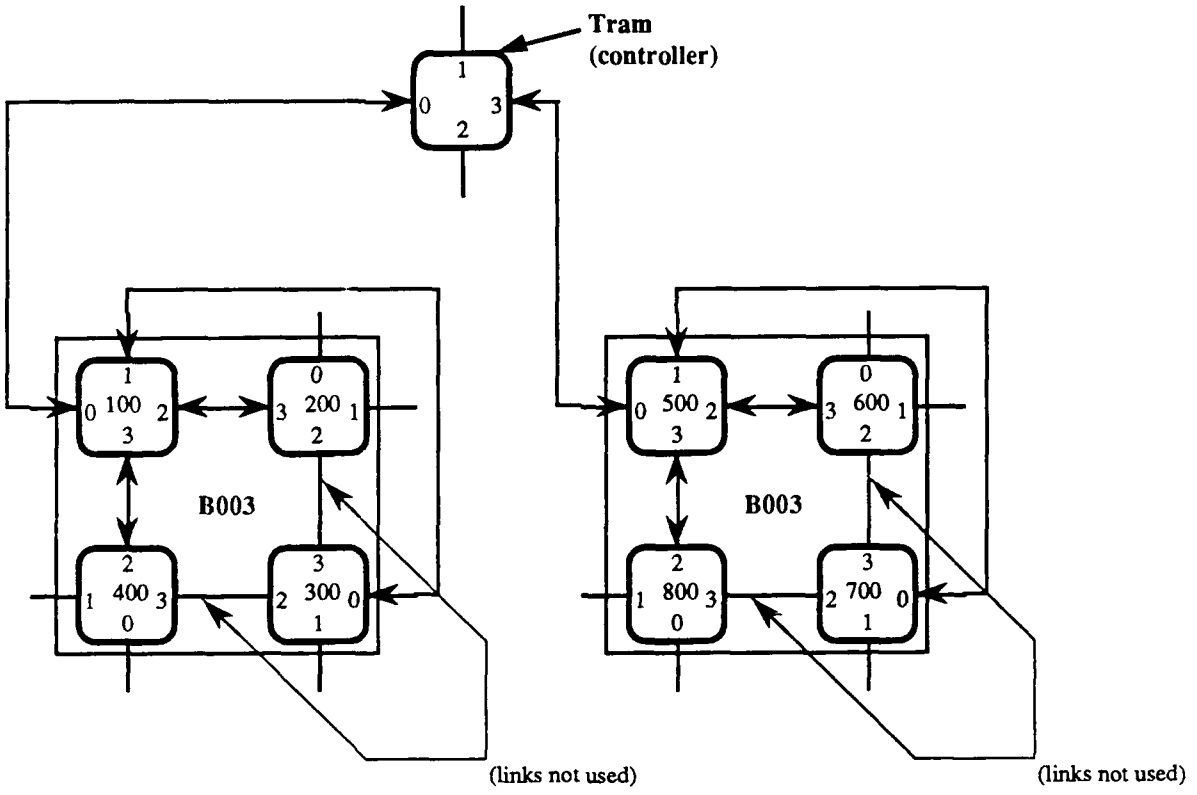


Figure 3.3 **Tree Model Physical Link Diagram**

3. Loop

The loop topology is a direct derivative of the linear model. The data flow is unidirectional instead of bidirectional as in linear model. Figure 3.4 depicts the loop model in a block diagram. Workpackets flow from the controller to the network. The results continue to flow in the same direction as workpackets and are transmitted by the last worker in the chain back to the controller via a separate link. Figure 3.5 is a proposed hardware connection of the T800 transputers.

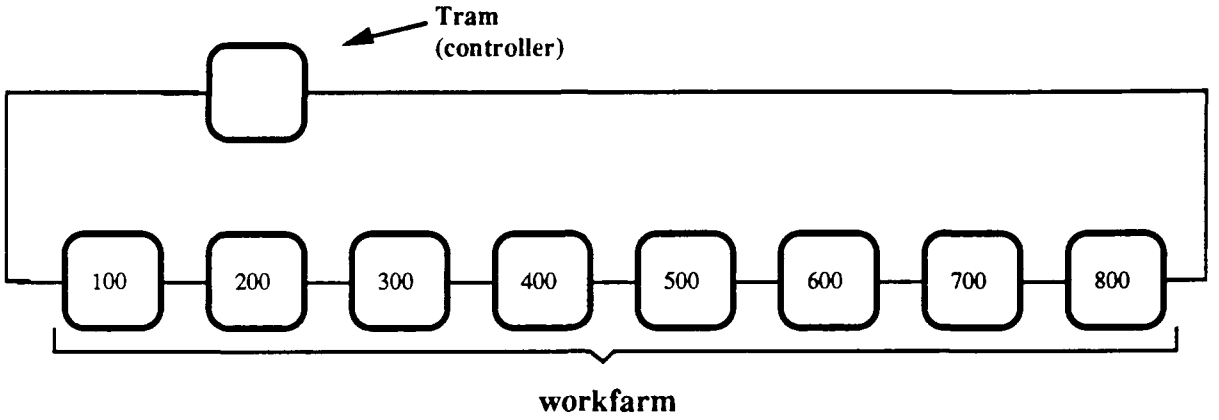


Figure 3.4 **Loop Model Block Diagram**

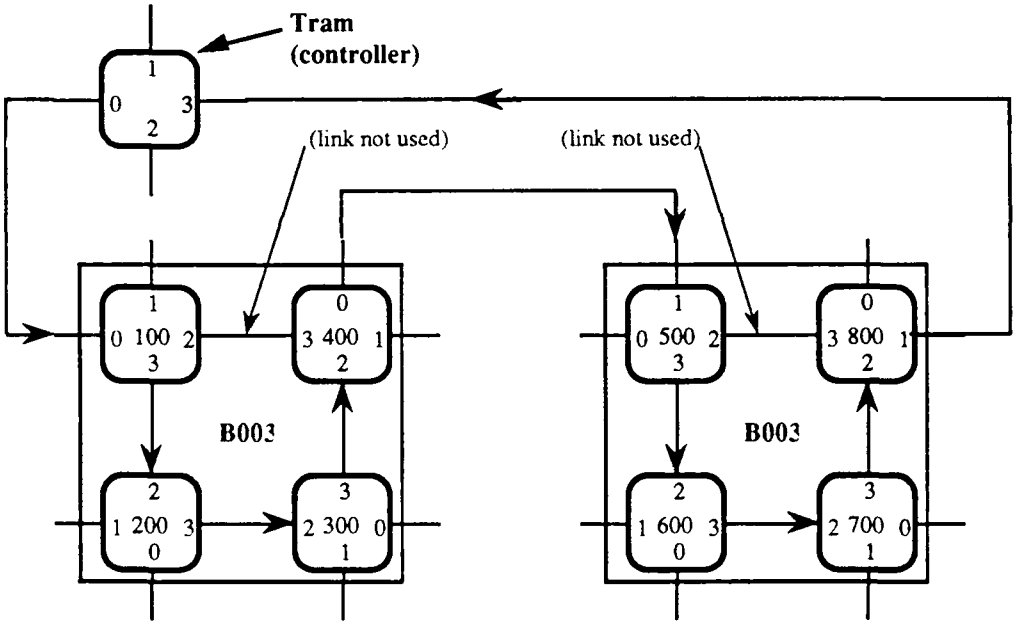


Figure 3.5 **Loop Model Physical Link Diagram**

4. Cube

The cube is the most complicated of the four models. It combines features of the tree and the loop topologies. Figure 3.6 is an

abstract diagram of the nodes and the link connections. Output from the controller would flow to the first worker where the workpackets (and results) could be forwarded to one of the three adjacent nodes along three different channels depending on availability. Each of these three adjacent nodes could in turn forward the workpackets (and results) to two other nodes, as shown in the figure. Ultimately all results will converge on the last worker in the cube which sends the results on to the controller via a separate link.

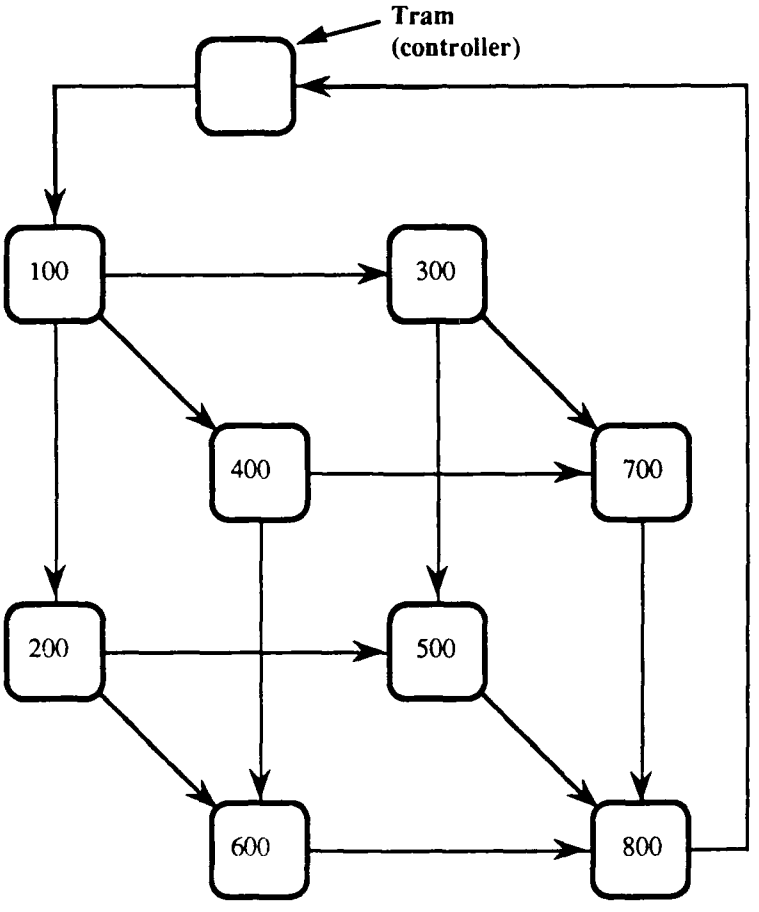


Figure 3.6 **Cube Model Block Diagram**

C. Algorithm

1. General Algorithm Structure for All Topologies

As mentioned in the previous section, there are several possible topologies in organizing a workfarm, e.g., linear, loop, tree, cube, and many others. The linear and tree topologies were investigated in this project. To test these topologies, an algorithm was designed to produce independently processable workpackets. Each topology employs a similar set of processes to accomplish the task. The processes implemented may differ slightly due to the communication requirements in the different topologies. Process names will be in bold print for the remainder of this paper. Common to each of the workfarms is the **controller**. The purpose of the **controller** is to coordinate the various processes necessary to 1) generate work based on the number of strips the graphics screen is subdivided into, 2) send the work out to the network of transputers for processing, 3) receive the results from the network and 4) monitor the time it takes to process each batch of work. A general model of the **controller** is shown in Figure 3.7.

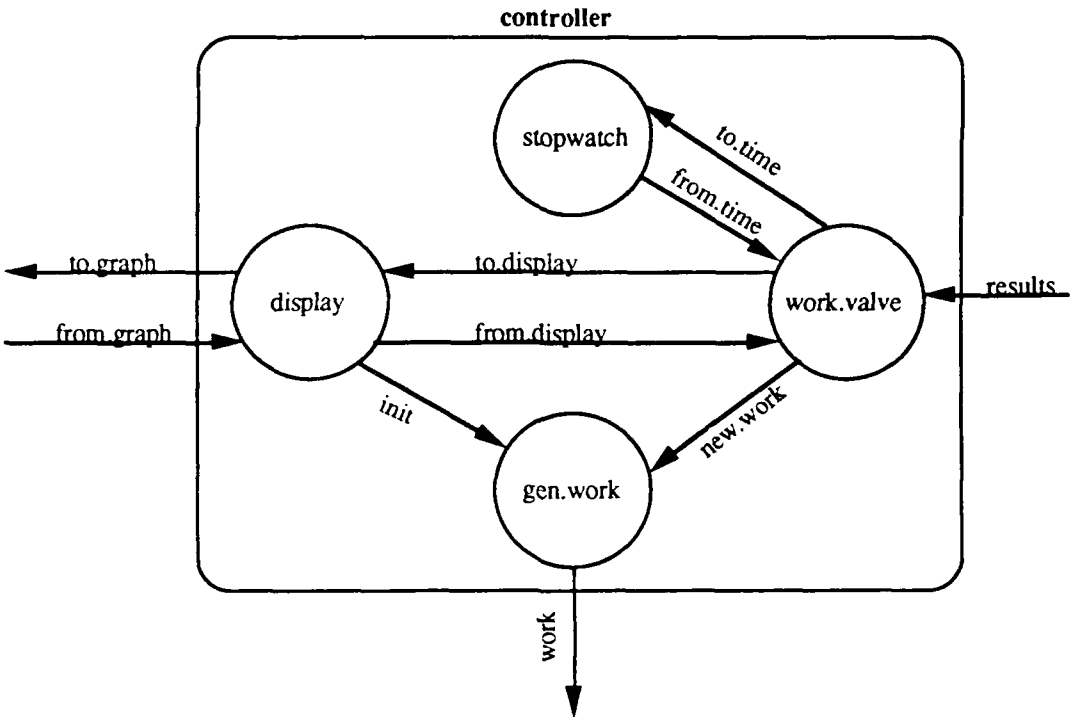


Figure 3.7 **Linear Model Controller Process**

The complete boxed diagram represents the **controller** process. This is loaded specifically into the root transputer. Each of the circled processes are part of the overall process and they run in virtual parallelism. In other words, they are not truly running in parallel. The four processes are time sliced based on their priority levels. Since they are all specified to run at high priority, they each receive equal processing time. Therefore, no one process receives priority over any other.

The arrowed lines between processes indicate designated software channels between those processes along which data may be passed. Arrows extending beyond the bounds of the **controller** signify

channels being routed along physical links to and from other hardware. The work and results channels are used to communicate with the network of workers while the to.graph and from.graph channels are used for sending and receiving data between the display and external filing hardware.

2. Linear Algorithm

The workfarm is where the workpackets are processed. The process which performs the calculations is consistent in each worker and in each topology. Any differences in the algorithm loaded into the network workers are due to communications requirements which depend on the specific topology being employed. It is also important to note that since more than one layer of workers is being used in every workfarm, physical links must be established between upper level workers and lower level workers in the hierarchy so that workpackets may be communicated to all workers in the farm. The process to be loaded into the linear workfarm is **pixel.gen** and is shown in Figure 3.8.

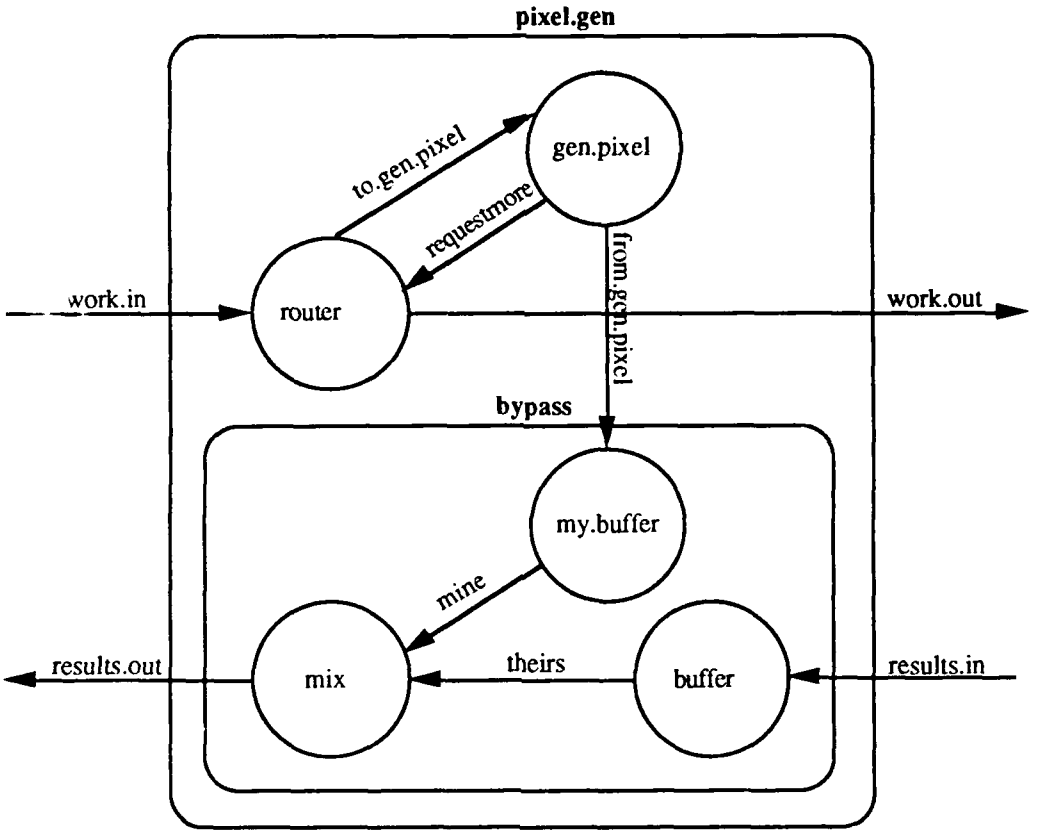


Figure 3.8 **Linear Model Pixel.gen Process**

The channels labelled work.in, work.out, results.in and results.out are along physical links and they are connected either to the TRAM (in the case of the first worker in the farm) or to other adjacent workers. The same discussion for processes internal to **controller** apply to the process **pixel.gen**. Note that **pixel.gen** has two levels of subprocesses. Processes **router**, **gen.pixel** and **bypass** all are at the same level while processes **my.buffer**, **buffer** and **mix** are all one level deeper.

The linear workfarm is implemented in two schemes. The first scheme is a non-addressed mode in which the **controller** sends

workpackets out to the network without specifying which worker is going to process the workpacket. The second scheme uses an addressed mode in which each workpacket contains the destination worker's unique address within the array. This allows a new workpacket to go directly to the worker that has returned a completed workpacket rather than to an opportunistic worker closer to the controller as the non-addressed mode would allow.

The basic operation of the algorithm is as follows. **Controller** contains **work.valve**, **display**, **gen.work** and **stopwatch**. These four sub-processes are running in parallel within **controller**. **Display** initializes the network of workers, via **gen.work**, with initialization data necessary to carry out the processing of the workpackets (initialization phase). This is sent from the **controller** via the channel annotated work.out. This output is fed to each of the workers by entering the first worker in line via its input channel work.in. Once a worker has been initialized with the appropriate data it will signal back to **work.valve** to send workpackets. Each worker will first get one workpacket to be processed by **gen.pixel**, then will fill its buffer (**store**, within **router**, is not visible in these figures) to the limit set by the variable buffer.size. When this task has been completed, this worker is full and any more initial workpackets will be forwarded to the next worker in line via channel work.out from **router** and channel work.in of the next worker. This process is repeated until all of the initial batch of workpackets has been sent to the network. Since the linear model has been implemented using two schemes, non-addressed and

addressed, they have different methods for sending workpackets out to the array of workers. A integer value in **display**, called address, is set to either 999 or some other number. When address is set to 999, the non-address mode is used for workpacket distribution. Otherwise, workpacket distribution is done according to the address mode. During the distribution of the initial batch of workpackets for the non-addressed mode, a workpacket is shipped without regards to which worker will receive it. The first worker with an available buffer opening will absorb that workpacket. The number of workpackets to be sent out during this phase equals the number of workers in the array multiplied by the buffersize plus one. With a standard number of eight workers in the array, the number of workpackets initially sent would be:

$$\#\text{workpackets} = 8*(10+1) = 88 \quad \text{where:} \left\{ \begin{array}{c} 8 = \text{number of workers} \\ 10 = \text{possible buffersize} \\ 1 = \text{workpacket for processing} \end{array} \right\}$$

In the non-address mode, it is conceivable that the first couple of workers will begin processing their buffered packets prior to all of the workers receiving an equal percentage of the initial allotment. Therefore, until processing overtakes the first processors, they may accept larger percentages of the initial workpacket allotment. As soon as the first processor completes a packet, it will return the results via **my.buffer** and **mix**. The results are then passed from **pixel.gen** to **controller** via the channels results.out and results.1 respectively. Once in **controller**, **work.valve** will send the results on to **display**.

In the addressed mode, **work.valve** sends the initial workpackets to the workfarm deterministically. **Work.valve** sends the initial workpackets as bundles to each worker in the array so every worker receives exactly the same initial number of workpackets. The number of workpackets in each bundle is the same and is equal to adding one to the current buffersize, i.e.,

$$\#\text{workpackets/worker} = 10+1 = 11 \quad \text{where: } \left\{ \begin{array}{c} 10 = \text{possible buffersize} \\ 1 = \text{workpacket for processing} \end{array} \right\}$$

Consequently with eight workers in the workfarm the total number of packets shipped initially in this example would be 88. The major difference here is that each worker will get the same number of initial packets.

The elapsed time for each batch of work was accomplished by triggering **stop.watch** to get a start time at the end of the initialization phase. Then after **work.valve** receives the results from the last workpacket, **stop.watch** is triggered again to get the stop time. The elapsed time is the difference between the stop time and the start time and represents the total amount of time taken to process all of the workpackets after the initialization phase for the network.

Several parameters are important in this workfarm. One of the most important parameters is the variable stripsize. One goal might be to section a monitor screen up into parts (or strips) and then randomly generate a color for each pixel (one byte for each pixel) of that strip. Reducing the stripsize reduces the number of pixels (or bytes) that need to be calculated in that strip. Consequently the

processing time for a strip decreases along with stripsize. However, the other side of this tradeoff is that as the stripsize shrinks, the maximum number of strips required to complete the screen increases. These two variables are inversely proportional. In order to calculate the color for the screen randomly, a random number generation library routine is called. It is actually called twice, once in **gen.work** and then again in **gen.pixel**. Both results are used in **gen.pixel**. The value calculated by **gen.pixel** is compared against the value forwarded by **gen.work**. If the former is not within plus or minus the window.width of the latter, then **gen.pixel** recalculates a random number and does the comparison again. This procedure causes each workpacket to be processed for a different period of time. Once a match has been made within the constraint of the window.width, the random number created by **gen.pixel** is returned with the results. The reason for replacing the random number value with the one calculated in **gen.pixel** is to scale the random number to a value between 0 and 255. Since there are only 256 possible colors per pixel, the calls produced by **gen.pixel** must produce results within the range of 0 to 255. Obviously as the window.width requirement becomes more constrained, the longer the process will take to converge and processing time will increase.

Every run of the workfarm decreases the stripsize by one half. The maximum stripsize is 128 bytes and the minimum stripsize is 1 byte. Each reduction in stripsize effectively doubles the maximum workload. It also reduces the amount of processing time for each

workpacket because each successively smaller stripsize contains only half as many bytes as the previously larger stripsize. By reducing the processing time per workpacket, the rate at which workpackets and results will be transmitted increases. It is possible that the first few workers will become a communications bottleneck while attempting to cope with the demands of more distal workers. This is because more distal workers have less of a requirement for communications due to their positioning within the network.

3. Tree Algorithm

The tree topology required some additional communication considerations that were not required in the linear model. Since the tree topology is a branching structure, allowances had to be made for communications to follow these branches freely in the non-addressed mode. Workpackets need to be able to go to any worker that is available. The algorithm was basically the same as the **pixel.gen** used in the linear algorithm with additional branching communication channels. The controller was modified to handle two output channels and two input channels to the network as shown in Figure 3.9. A new workpacket was sent to the specific branch on which the last results packet was received.

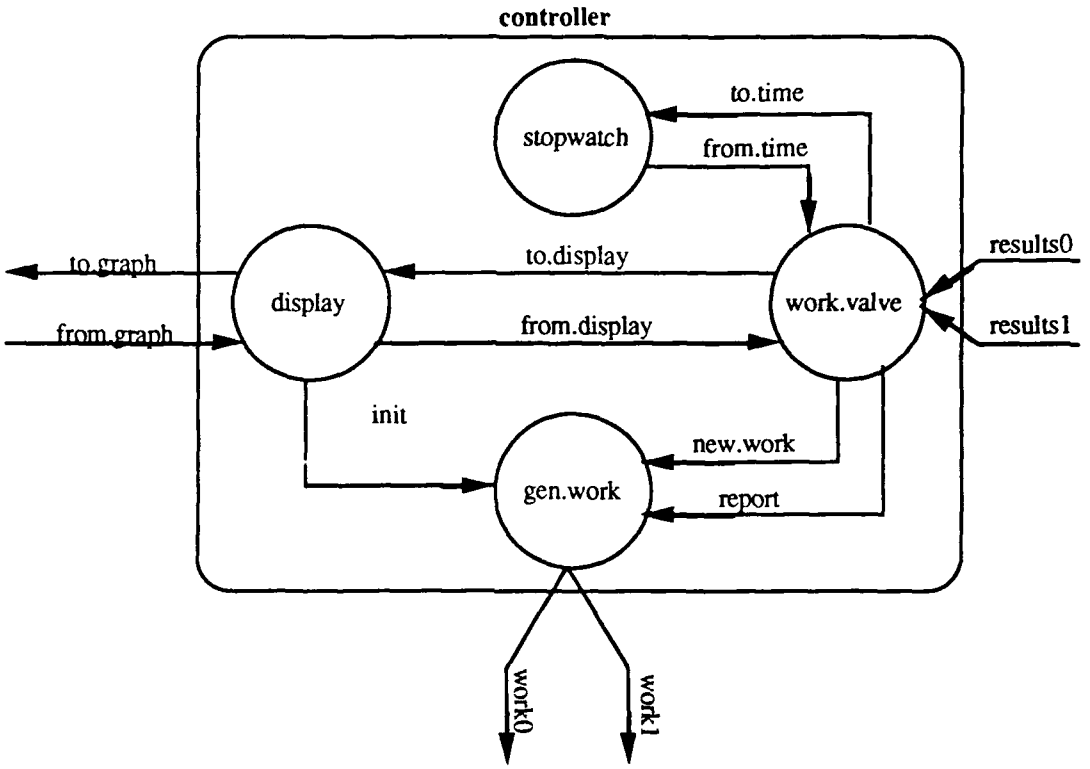


Figure 3.9 **Tree Model Controller Process**

There were two modified versions of **pixel.gen**. The first, **pixel.gen2**, was modified to handle three input and output channels as required by the second level workers. In addition, **pixel.gen2** was also modified to allow a second level worker to reroute any workpacket among the third level workers in the non-addressed mode. This allowed a workpacket to be cycled through the third level workers until one that could process or buffer it was found. Figure 3.10 shows **pixel.gen2** with the channel, reroute, which allows the rerouting of the workpackets as just discussed.

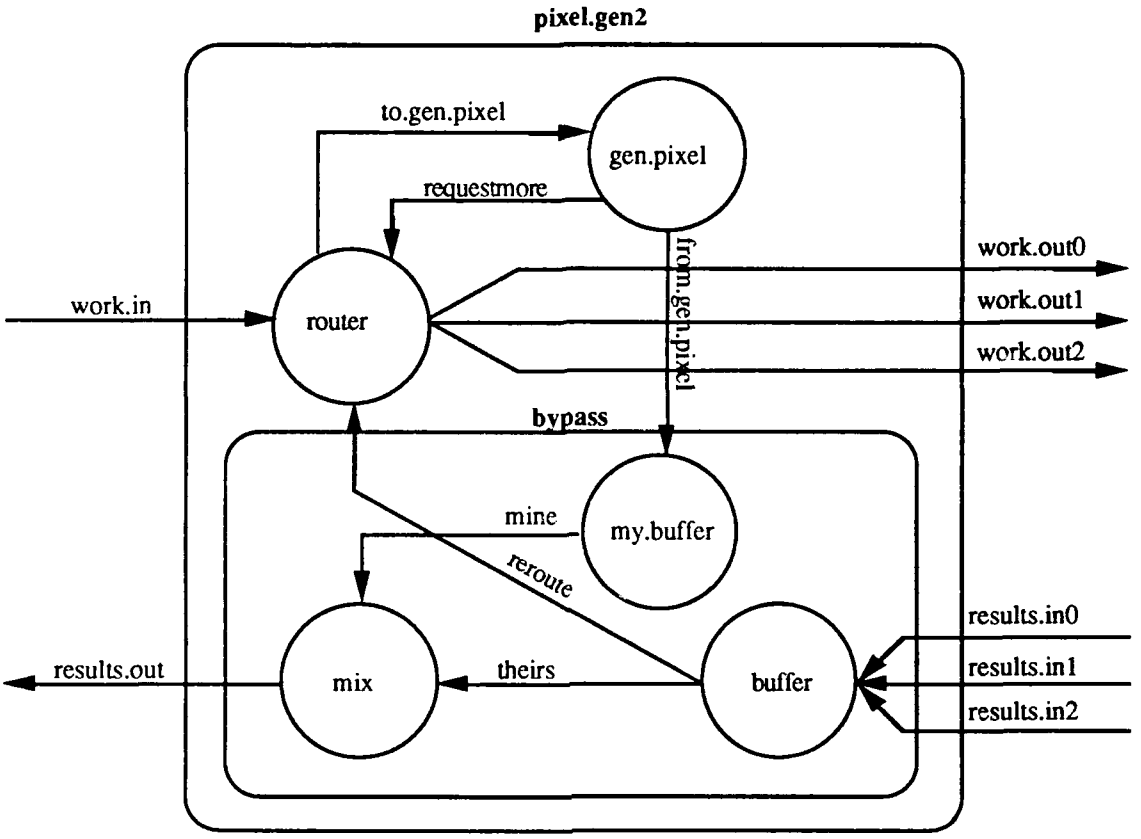


Figure 3.10 **Tree Model Second Level Worker Pixel.gen2 Process**

The third level workers did not require any additional output channels for branched workers and was not modified to handle this. If additional workers were to be appended to the third level workers, then **pixel.gen3** could be modified the same as **pixel.gen2** to allow for communications. **Pixel.gen3** does include a channel, reroute, which will send an unprocessed workpacket back up to the second level worker for redistribution to another third level worker. **Pixel.gen3** is shown in Figure 3.11.

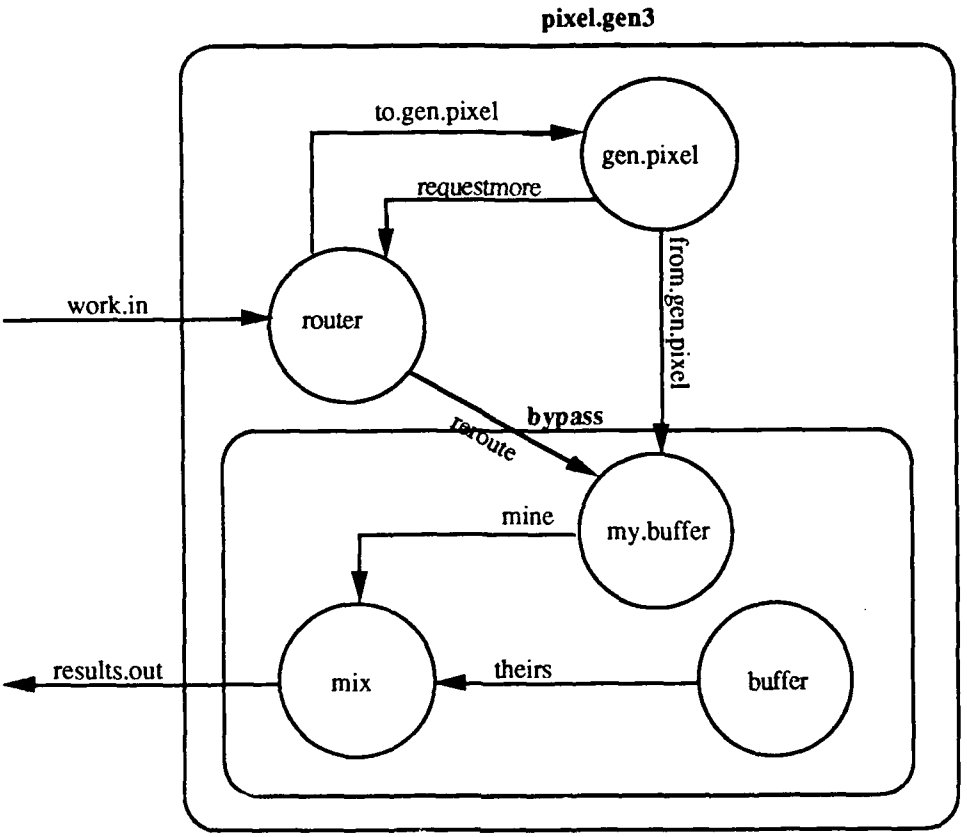


Figure 3.11 **Tree Model Third Level Pixel.gen3 Process**

IV. RESULTS

A. Linear Topology

The load balancing results [Ref. 5], produced by varying the stripsize and buffersize parameters, are graphically depicted in Appendix B and Appendix C. Appendix B contains the results for the non-addressed mode operation of the linear topology. Appendix C contains the results for the addressed mode operation. The graphs are broken down by stripsize with workpackets done per worker versus buffersize. Examining these graphs shows a wide range of load balancing. Large stripsizes corresponding to larger workpacket sizes resulted in more even load balancing for both the addressed and non-addressed modes of operation. For the large stripsizes, buffersize did not seem to have a significant effect on the load balancing. As the stripsize was decreased below 32, the load balancing became less symmetric with the most distal workers in the network taking on less work with small buffersizes. The load balancing appears to smooth out as the buffersize increases with stripsizes down through 8, but careful inspection of the actual loads shows that the further the worker is from the controller, the greater the load it will carry in terms of workpackets processed. This can be attributed to the decreased processing time associated with smaller workpackets. Decreasing the workpacket size reduces the processing time required by any given worker. This results in a communications buildup that increases the more proximal to the controller a worker is. Those workers closest to

the controller become inundated with communication demands to serve workpackets and receive results from the most distal workers because of the faster processing time due to smaller packets. For the non-addressed mode, serving workpackets to the network that are smaller than 16 bytes in length results in severely unbalanced loading.

The addressed mode provides a slight advantage over the non-addressed mode in that there is a point of convergence for load balancing given any stripsize. A buffersize of 1 produced a reasonable load balance for all workers in the network regardless of stripsize. Distributing workpackets by request to the workers, vice the first available method used in the non-addressed mode, resulted in the pattern seen in Appendix C.

B. Tree Topology

Due to the bilateral topology of the tree, the load balancing results, as shown in Appendix D and Appendix E, were as expected for large values of stripsize. The load was shared evenly between both main branches of the tree. In addition the tertiary level workers shared identical loads within a single branch. The tertiary workers in one main branch also handled exactly the same amount of work as the tertiary level workers in the other main branch. Both workers in the second level handled the same amount of work, but processed fewer packets than any given tertiary worker. This was expected as their position in the network dictated their task in mediating communications between the controller and the tertiary workers.

Both the non-addressed and addressed modes responded similarly to the decrease in stripsize. The load balancing remained symmetric through a stripsize of sixteen at which point the load balancing became asymmetrical. As a result of faster processing time for smaller packets, whichever branch gained control of the initial workpackets first would continue to absorb most of them for the remainder of a batch of work.

C. Comparison of Topology Performance

In order to do a comparative analysis of the two topologies, timing results were obtained for both the linear and tree models for both operating modes over a range of workpacket sizes. Table 4.1 contains sample results for each of these categories. These figures were obtained for a buffersize equal to 2 in all cases. In comparing the linear with the tree for any given mode and stripsize, it is clear that the tree has a temporal advantage in processing a complete batch of work. Distributing work to the network deterministically using the addressed mode resulted in quicker processing times over the non-addressed mode. This was due to less processing time being required to determine if a worker could handle the work. The deterministic method merely checked to see if a worker was the destination worker. If not, the workpacket was passed on. For example, with a large stripsize (128 bytes), the tree topology using the addressed mode is 0.013273 seconds faster than the linear model using the non-address mode.

Topology	Stripsize	Non-Addressed (s)	Addressed (s)
Linear	128	1.606285	1.605210
	64	1.725431	1.724606
	32	1.970065	1.969109
	16	2.511180	2.479759
	8	3.700452	3.626762
Tree	128	1.597760	1.593012
	64	1.653389	1.643526
	32	1.761103	1.750702
	16	1.971369	1.970354
	8	3.277614	3.222384

Table 4.1 **Comparison of Linear Model and Tree Model Timing Results**

Although the linear model was simpler to set up, the tree topology did have some advantages which might make it more useful. In terms of timing, the tree topology was faster than the linear model. For many applications the difference in speed between the two topologies may not be of any consequence to the user. However, for applications where every additional margin of speed is of the essence, a tree type model may prove advantageous by creating the shortest communication distances, with the fewest intermediate nodes between a root controller and the most distal workers in the network. The linear model could potentially become bogged down in communication delays with increasing numbers of workers arranged in that topology.

Load balancing considerations make the tree a better model for processor utilization. There was a greater percentage of processors in the tree network doing useful processing as compared with the linear

model. This was particularly evident as the stripsize decreased below thirty-two.

V. CONCLUSIONS and DISCUSSION

Difficulties encountered were centered around debugging. The Transputer Development System, TDS, that was used did not provide any debugging facilities. Therefore, attempting to localize bugs in the network was a time consuming and not altogether enjoyable task. The controller level of each of the networks was simpler to debug because it interfaced directly with the monitor controlling boards. However, the network of workers was essentially invisible to debugging except at the source code level. Possible alternatives to working in the Transputer Development System would be using the Profiler [Ref. 6] and NDB (network debugger) [Ref. 7] produced by Parasoft Corporation. The Profiler is a performance monitor composed of an Execution Profiler, a Communication Profiler, and an Event Profiler. The Execution Profiler monitors time spent in individual routines, the Communication Profiler evaluates the time spent in communications and I/O, and the Event Profiler shows interactions between processors and allows user-specified events to be monitored. NDB is a symbolic source and assembly level debugger for parallel computers. Using this tool it is possible to determine how far a program has run. The problem of not being able to completely monitor the network will still be a problem though.

The tree network, running in the non-addressing mode, did not execute any batches of work beyond the first two within a stripsize of eight. Due to time constraints and the inherent difficulties with debugging, this problem remains to be resolved. Data from this

configuration with stripsizes ranging from 128 to 16 as compared to data from the other network configurations appeared to be accurate and was considered useful.

The transputer has an inherent simplicity based on the CSP philosophy which makes it a valuable tool for workfarm research. Many topologies, such as a cube or loop, can be investigated quickly and simply using this device. Additional research should be investigated utilizing a shared memory schemes to enhance the utility of this device in problems requiring shared databases. In addition more effort should be applied towards the production of debugging tools for multiprocessor networks.

Another approach to debugging small networks of transputers, i.e., less than 16, could lie in the CSP design philosophy of the transputer itself. A debug specific hardware board could be designed which taps one link to each transputer in the network thus enabling a debug program to monitor each of the processes executing in any or all of the transputers in the network connected to the debug hardware. Obviously this would defeat the connectivity of each of the transputers in the network to some degree. However, in the interest of providing more debugging capability and hence more robust programs, this may be an area worth investigating.

APPENDIX A

DETAILED SOURCE CODE - LINEAR TOPOLOGY

```
-- link definitions
VAL  link0out  IS  0:
VAL  link1out  IS  1:
VAL  link2out  IS  2:
VAL  link3out  IS  3:
VAL  link0in   IS  4:
VAL  link1in   IS  5:
VAL  link2in   IS  6:
VAL  link3in   IS  7:

-- declarations
VAL  numT8     IS  8:
VAL  numT4     IS  0 :
VAL  numTs     IS  numT8+numT4:
-- channel declarations
CHAN OF ANY to.graph,from.graph :
CHAN OF ANY graph.to.mouse,mouse.to.graph :
CHAN OF ANY to.net  ,from.net   :
CHAN OF ANY   :
CHAN OF ANY to.time,from.time:
[numT8+numT4] CHAN OF ANY results:
[numT8+numT4] CHAN OF ANY work  :
--   VAL assignments
VAL  work.in       IS  [4,6,6,6,5,6,6,6]:
VAL  work.out      IS  [3,3,3,0,3,3,3,1]:
VAL  results.in    IS  [7,7,7,4,7,7,7,5]:
VAL  results.out   IS  [0,2,2,2,1,2,2,2]:

--------------------------------------------------------------------------
PROC controller (CHAN OF ANY to.graph,from.graph,work,results)
--Process which provides control for workfarm.  Contains four internal
--parallel processes; PROC work.valve, PROC gen.work, PROC display, and
PROC stop.watch.
--------------------------------------------------------------------------
  #USE "raycom.tsr"
  #USE "\misclib\grafsymb.tsr"
  -- declarations
  CHAN OF ANY new.work,init,to.display,from.display:
  CHAN OF ANY to.time,from.time:
  CHAN OF ANY report:

  ------------------------------------------------------------------------
  PROC work.valve (CHAN OF ANY results,new.work,
                   to.time,from.time,to.display,from.display,report)
  --PROC work.valve monitors the network for incoming results which it
  --sends to PROC display.  At the beginning and end of each batch of
  --work, PROC work.valve gets the current time from PROC stop.watch.
  --At the end of a batch of work, PROC work.valve computes the elapsed
  --time the network took to process that batch of work
```

```
-- declarations
#USE "raycom.tsr"
INT32 seed:
INT64 compares:
INT command,maxwork:
INT num,packets,buffer.size:
INT time,lastT:
INT address,j:
INT x,y:
INT rmax:
INT stripsize:
INT randnum,workdone:
INT window.width:
INT iters:
[128] BYTE pixels:
BOOL active:

SEQ
  active := TRUE
  WHILE active
    ALT
      results ? command
        IF
          command = c.result
            SEQ
              results ? x;y;[pixels FROM 0 FOR stripsize];address
              to.display ! c.result;x;y;
                           [pixels FROM 0 FOR stripsize]
              new.work ! address
              workdone := workdone + 1
              IF
                workdone = maxwork
                  SEQ
                    to.time ! TRUE
                    report ! TRUE
                ELSE
                  SKIP
          command = c.report
            SEQ
              from.time ? time
              to.display ! c.time;time
          command = c.report.data
            SEQ
              results ? packets;compares;num
              to.display ! c.report.data;packets;compares;num
          command = c.init
            SEQ
              results ? seed;window.width;rmax;
                        stripsize;lastT;buffer.size
              to.time ! TRUE            -- start timing
              IF
                address = 999
                  SEQ i = 0 FOR iters
                    new.work ! address
```

```
                TRUE
                  SEQ i = 0 FOR iters/(buffer.size+1)
                    SEQ j = 0 FOR buffer.size
                      new.work ! (i+1)*100
            command = c.test
              SEQ
                results ? command
                to.display ! c.test;command
            TRUE
              SKIP
        -- initialization sequence
        from.display ? stripsize;lastT;address;buffer.size
          SEQ
            iters := lastT TIMES (buffer.size+1)
            maxwork := (512*512)/stripsize
            workdone := 0
:
--------------------------------------------------------------------------
PROC gen.work (CHAN OF ANY in,out,init,report)
--PROC gen.work creates new work as dictated by PROC work.valve.  Work
--is only sent to the network as results are returned.  The one
--exception to this is the initialization of the network for each
--batch of work.  At that time PROC display provides the initial
--values for the parameters to be used by the network in doing its
--calculations. PROC display sends this information to PROC gen.work
--which then sends it to the network.  When a reply is received by
--PROC work.valve that the network has been initialized then a trigger
--is sent to PROC gen.work to send a specific number of workpackets to
--the network.  As results are received by PROC work.valve, PROC
--gen.work sends new workpackets.
--------------------------------------------------------------------------
  -- declarations
  #USE "\tdsiolib\fpmath8.tsr"
  #USE "raycom.tsr"
  INT32 seed,Tseed:
  INT x,y,lastT,buffer.size:
  INT address:
  INT maxrand:
  INT stripsize:
  INT randnum:
  INT window.width:
  INT maxwork,workdone:
  [128] BYTE pixels:
  BOOL active,trigger:
  --------------------------------------------------------------------------
  PROC random (INT rnum,VAL INT rmax)
  --PROC random is a call to a library process which generates a
  --random number.
  --------------------------------------------------------------------------
    REAL32 result:
    SEQ
      RANP (result,seed)
      rnum := INT ROUND (result*(REAL32 ROUND rmax))
  :
```

```
  SEQ
    active := TRUE
    seed := 245786 (INT32)
    WHILE active
      ALT
        in ? address
          IF
            workdone < maxwork
              SEQ
                random (randnum,maxrand)
                out ! c.ray;x;y;randnum;address
                workdone := workdone + 1
                x := x + stripsize
                IF
                  x > 511
                    SEQ
                      y := y + 1
                      x := 0
                  TRUE
                    SKIP
            TRUE
              SKIP
        init ? Tseed;window.width;maxrand;stripsize;lastT;buffer.size
          SEQ
            out ! c.init;Tseed;window.width;maxrand;stripsize;
                lastT;buffer.size
            maxwork := (512*512)/stripsize
            x := 0
            y := 0
            workdone := 0
        report ? trigger
          out ! c.report
:
--------------------------------------------------------------------------
PROC display (CHAN OF ANY in,out,to.graph,from.graph,init)
--PROC display generates the initial data required by the network to
--process the workpackets.  It sends this data to PROC gen.work which
--in turn sends the initialization data to the network.  PROC display
--also receives incoming results from PROC work.valve and sends them
--to a B007 graphics controller which displays the results on a color
--monitor.
--------------------------------------------------------------------------
  INT reply:
  INT64 compares:
  INT num,packets,total:
  INT maxwork,reports,time,buffer.size:
  INT x,y,command:
  INT address,j,k,l:
  INT x.pos,y.pos:
  INT mode,len:
  INT window.width,rmax,stripsize,lastT:
  [4]INT params:
  [128]BYTE pixels:
  BOOL m.l,m.m,m.r:
  BOOL active,not.done:
```

```
VAL seed IS 127463 (INT32):
--------------------------------------------------------------------------
PROC set.scroll (VAL INT top,bottom)
--------------------------------------------------------------------------
  SEQ
    mode := 0
    params[0] := VT220.set.scroll
    params[1] := top
    params[2] := bottom
    to.graph ! c.to.VT220;mode;params
:
--------------------------------------------------------------------------
PROC clear.screen ()
--------------------------------------------------------------------------
  SEQ
    mode := 0
    params[0] := VT220.clear.screen
    to.graph ! c.to.VT220;mode;params
:
--------------------------------------------------------------------------
PROC tab ()
--------------------------------------------------------------------------
  SEQ
    mode := 0
    params[0] := VT220.tab
    to.graph ! c.to.VT220;mode;params
:
--------------------------------------------------------------------------
PROC c.return ()
--------------------------------------------------------------------------
  SEQ
    mode := 0
    params[0] := VT220.return
    to.graph ! c.to.VT220;mode;params
:
--------------------------------------------------------------------------
PROC print.screen ()
--------------------------------------------------------------------------
  SEQ
    mode := 0
    params[0] := VT220.print.screen
    to.graph ! c.to.VT220;mode;params
:
--------------------------------------------------------------------------
PROC write.num (VAL INT num)
--------------------------------------------------------------------------
  SEQ
    mode := 1
    params[0] := VT220.num
    params[1] := num
    to.graph ! c.to.VT220;mode;params
:
--------------------------------------------------------------------------
PROC write.num.xy (VAL INT num,x,y)
--------------------------------------------------------------------------
```

```
  SEQ
    mode := 1
    params[0] := VT220.num.xy
    params[1] := num
    params[2] := x
    params[3] := y
    to.graph ! c.to.VT220;mode;params
:
--------------------------------------------------------------------------
PROC write.text (VAL [] BYTE text)
--------------------------------------------------------------------------
  SEQ
    mode := 2
    params[0] := VT220.text
    len := SIZE text
    to.graph ! c.to.VT220;mode;params;len;text
:
--------------------------------------------------------------------------
PROC write.text.xy (VAL [] BYTE text,VAL INT x,y)
--------------------------------------------------------------------------
  SEQ
    mode := 2
    params[0] := VT220.text.xy
    params[2] := x
    params[3] := y
    len := SIZE text
    to.graph ! c.to.VT220;mode;params;len;text
:
--------------------------------------------------------------------------
PROC highlight ()
--------------------------------------------------------------------------
  SEQ
    mode := 0
    params[0] := VT220.highlight
    to.graph ! c.to.VT220;mode;params
:
--------------------------------------------------------------------------
PROC underline ()
--------------------------------------------------------------------------
  SEQ
    mode := 0
    params[0] := VT220.underline
    to.graph ! c.to.VT220;mode;params
:
--------------------------------------------------------------------------
PROC wait.for.click()
--------------------------------------------------------------------------

  SEQ
    m.r := FALSE
    m.m := FALSE
    m.l := FALSE
    to.graph ! c.get.mouse
    from.graph ? x.pos;y.pos;m.l;m.m;m.r
    WHILE (NOT m.r) AND (NOT m.m) AND (NOT m.l)
```

```
      SEQ
        to.graph ! c.get.mouse
        from.graph ? x.pos;y.pos;m.l;m.m;m.r
:

SEQ
  active := TRUE
  window.width := 1000
  lastT := 8
  rmax := 1000
  stripsize := 128
  address := 0
  j := 0
  k := 0
  l := 0
  WHILE (stripsize >= 1)
    SEQ
      SEQ buffer.size = 1 FOR 20
        SEQ
          -- color monitor
          to.graph   ! c.hide.cursor
          from.graph ? reply
          to.graph   ! c.init.crt;sony
          from.graph ? reply
          to.graph   ! c.select.screen;0
          from.graph ? reply
          to.graph   ! c.clear.screen;0
          from.graph ? reply
          to.graph   ! c.display.screen;0
          from.graph ? reply
          to.graph   ! c.select.colour.table;0
          from.graph ? reply
          -- VT220
          clear.screen ()
          set.scroll (5,24)
          highlight()
          IF
            address = 999
              write.text.xy ("Work Farm - Linear/no address
                              mode",1,20)
            TRUE
              write.text.xy ("Work Farm - Linear/address mode",1,20)
          highlight()

          to.graph ! c.to.host;window.width
          to.graph ! c.to.host;stripsize
          to.graph ! c.to.host;buffer.size

          write.text.xy ("Window.width: ",3,1)
          write.num.xy  (window.width,3,12)
          write.text.xy ("# Transputers: ",3,50)
          write.num.xy  (lastT,3,65)
          write.text.xy ("Random number range 0 - ",4,1)
          write.num.xy  (rmax,4,25)
          write.text.xy ("Stripsize: ",4,50)
```

```
write.num.xy  (stripsize,4,61)
write.text.xy ("Buffer size: ",5,50)
write.num.xy  (buffer.size,5,63)
c.return ()
underline ()
write.text ("Transputer #   Work Done    Compares")
underline ()
c.return ()
c.return ()

out ! stripsize;lastT;address;buffer.size
init ! seed;window.width;rmax;stripsize;lastT;buffer.size
reports := 0
not.done := TRUE
maxwork := 0
WHILE not.done
  SEQ
    in ? command
    IF
      command = c.result
        SEQ
          in ? x;y;[pixels FROM 0 FOR stripsize]
          to.graph ! c.mandelbrot;stripsize;x;y:
                     [pixels FROM 0 FOR
                      stripsize]
          maxwork := maxwork + 1
      command = c.report.data
        SEQ
          in ? packets;compares;num
          reports := reports + 1
          to.graph ! c.to.host;num
          to.graph ! c.to.host;packets
          to.graph ! c.to.host;(INT compares)

          write.text ("    ")
          write.num (num)
          tab ()
          IF
            num < 999
              tab ()
            TRUE
              SKIP
          write.num (packets)
          tab ()
          tab ()
          write.num (INT compares)
          c.return ()
      command = c.time
        in ? time
      command = c.test
        SEQ
          in ? command
          write.text ("Test ")
          write.num (command)
          c.return()
```

```
                    TRUE
                      SKIP
                  IF
                    reports = lastT
                      SEQ
                        to.graph ! c.to.host;maxwork
                        to.graph ! c.to.host;time

                        c.return ()
                        write.text ("Total work: ")
                        write.num (maxwork)
                        c.return ()
                        write.text ("Time (usec): ")
                        write.num (time)
                        c.return ()
                        not.done := FALSE
                    TRUE
                      SKIP
          stripsize := stripsize/2
  :
  ------------------------------------------------------------------------
  PROC stop.watch (CHAN OF ANY in,out)
  --PROC stop.watch provides timing information from the high priority
  --clock running in 1µsec ticks.  PROC stop.watch is called in this
  --algorithm by PROC work.valve
  ------------------------------------------------------------------------
    INT start,finish:
    TIMER clock:
    BOOL active,toggle:

    SEQ
      active := TRUE
      WHILE active
        PRI PAR
          SEQ
            in    ? toggle
            clock ? start
            in    ? toggle
            clock ? finish
            out   ! (finish-start)
          SKIP
  :

  PRI PAR
    PAR
      work.valve (results,new.work,to.time,from.time,
                  to.display,from.display,report)
      gen.work (new.work,work,init,report)
      display (to.display,from.display,to.graph,from.graph,init)
      stop.watch (to.time,from.time)
    SKIP
:

------------------------------------------------------------------------
PROC pixel.gen (CHAN OF ANY work.in,work.out,results.out,results.in,
```

```
                VAL INT mynum)
--PROC pixel.gen is run by each worker in the workfarm.  It contains
--processes which control the routing, processing and buffering of
--workpackets and processes which control the flow of results back up to
--the controller.
--------------------------------------------------------------------------
  #USE "raycom.tsr"    -- ray tracer command definitions

  -- declarations
  CHAN OF ANY to.ray.tr,from.ray.tr,requestmore: -- internal channels

  --------------------------------------------------------------------------
  PROC router (CHAN OF ANY work.in,work.out,to.ray.tr,requestmore,VAL
               INT mynum)
  --PROC router controls the flow of workpackets.  If a given worker in
  --the network cannot handle the current workpacket it is either
  --buffered or sent to another worker in the network.
  --------------------------------------------------------------------------
    -- declarations
    #USE "raycom.tsr"    -- ray tracer command definitions
    BOOL active :
    BOOL busy    :
    BOOL bufferfull:
    VAL buffmax  IS 20:
    VAL empty    IS -1:
    [buffmax] INT xtemp:
    [buffmax] INT ytemp:
    [buffmax] INT rantemp:
    [buffmax] INT addresstemp:
    INT address,buffer.size:
    INT  command,lastT :
    INT32  seed,myseed:
    INT window.width,rmax,stripsize:
    INT randnum:
    INT  packets.done:
    INT x,y:
    INT buffcount:
    BYTE sendmore:

    --------------------------------------------------------------------------
    PROC store (INT x,y,randnum,address)
    --PROC store  is the buffer for workpackes in any given worker.  If
    --the worker is not busy and PROC store has available workpackets,
    --one will be sent by PROC send to PROC gen.pixel
    --------------------------------------------------------------------------
      SEQ
        buffcount := buffcount + 1
        IF
          (buffcount = (buffer.size-1)) OR (buffcount = (buffmax-1))
            SEQ
              bufferfull := TRUE
              xtemp[buffcount] := x
              ytemp[buffcount] := y
              rantemp[buffcount] := randnum
              addresstemp[buffcount] := address
```

```
      TRUE
        SEQ
          xtemp[buffcount] := x
          ytemp[buffcount] := y
          rantemp[buffcount] := randnum
          addresstemp[buffcount] := address
:
--------------------------------------------------------------------------
PROC send ()
--PROC send removes workpackets from the workpacket buffer and
--routes them to PROC gen.pixel if PROC gen.pixel is not busy and
--the buffer has at least one workpacket.
--------------------------------------------------------------------------
  SEQ
    to.ray.tr ! c.ray;xtemp[buffcount];ytemp[buffcount];
                rantemp[buffcount];addresstemp[buffcount]
    bufferfull := FALSE
    packets.done := packets.done + 1
    IF
      buffcount > empty
        buffcount := buffcount - 1
      TRUE
        SKIP
:

SEQ
  active := TRUE
  WHILE active
    PRI ALT
      busy & requestmore ? sendmore
        IF
          buffcount = empty
            busy := FALSE
          TRUE
            send ()
      --(NOT busy) & requestmore ? sendmore
        --SKIP
      work.in ? command
        IF
          command = c.ray
            SEQ
              work.in ? x;y;randnum;address
              IF
                address = 999
                  IF
                    NOT busy
                      SEQ
                        to.ray.tr ! c.ray;x;y;randnum;address
                        packets.done := packets.done + 1
                        busy := TRUE
                    bufferfull
                      work.out ! c.ray;x;y;randnum;address
                  TRUE
                      store (x,y,randnum,address)
                address = mynum
```

```
                    IF
                      NOT busy
                        SEQ
                          to.ray.tr ! c.ray;x;y;randnum;address
                          packets.done := packets.done + 1
                          busy := TRUE
                      TRUE
                        store (x,y,randnum,address)
                  TRUE
                    work.out ! c.ray;x;y;randnum;address
            command = c.init
              SEQ
                -- initialization sequence
                work.in  ? seed;window.width;rmax;
                           stripsize;lastT;buffer.size
                myseed := seed
                seed := seed + 1231 (INT32)
                work.out  ! c.init;seed;window.width;rmax;stripsize;
                          lastT;buffer.size
                to.ray.tr ! c.init;myseed;window.width;rmax;stripsize
                packets.done := 0
                bufferfull := FALSE
                buffcount := empty
                busy := FALSE
            command = c.report
              SEQ
                work.out  ! c.report
                to.ray.tr ! c.report.data;packets.done
            command = c.test
              SEQ
                work.in ? command
                work.out ! c.test;command
            TRUE
              SKIP
:
--------------------------------------------------------------------------
PROC bypass (CHAN OF ANY results.in,results.out,from.ray.tr)
--PROC bypass receives results from PROC gen.pixel or from other
--workers
--downstream trying to send their results up to the controller   It
--provides a
--control point for the flow of results to the controller.
--------------------------------------------------------------------------
  -- declarations
  #USE "raycom.tsr"   -- ray tracer command definitions
  CHAN OF ANY mine,theirs :   -- internal channels

  --------------------------------------------------------------------------
  PROC my.buffer (CHAN OF ANY in,out)
  --PROC my.buffer receives results from PROC gen.pixel within that
  --specific worker and forwards that data on to PROC mix for
  --transmission to the controller
  --------------------------------------------------------------------------
    -- declarations
    #USE "raycom.tsr"   -- ray tracer command definitions
```

```
  INT command,address:
  INT x,y:
  INT32 seed:
  INT mynum,packets.done:
  INT64 compares:
  INT randnum,window.width,rmax,stripsize:
  [128] BYTE pixels:
  BOOL active:

  SEQ
    active := TRUE
    WHILE active
      SEQ
        in ? command
        IF
          command = c.result
            SEQ
              in  ? x;y;[pixels FROM 0 FOR stripsize];address
              out ! c.result;x;y;
                    [pixels FROM 0 FOR stripsize];address
          command = c.report.data
            SEQ
              in  ? packets.done;compares;mynum
              out ! c.report.data;packets.done;compares;mynum
          command = c.init
            in  ? stripsize
          command = c.test
            SEQ
              in ? command
              out ! c.test;command
          TRUE
            SKIP
:
--------------------------------------------------------------------------
PROC buffer (CHAN OF ANY in,out)
--PROC buffer receives results from downstream workers and forwards
--the results to PROC mix for transmission to the controller
--------------------------------------------------------------------------
  -- declarations
  #USE "raycom.tsr"   -- ray tracer command definitions
  INT command,lastT,buffer.size:
  INT x,y,address:
  INT32 seed:
  INT mynum,packets.done:
  INT64 compares:
  INT randnum,window.width,rmax,stripsize:
  [128] BYTE pixels:
  BOOL active:

  SEQ
    active := TRUE
    WHILE active
      SEQ
        in ? command
        IF
```

```
            command = c.result
              SEQ
                in  ? x;y;[pixels FROM 0 FOR stripsize];address
                out ! c.result;x;y;
                      [pixels FROM 0 FOR stripsize];address
            command = c.report.data
              SEQ
                in  ? packets.done;compares;mynum
                out ! c.report.data;packets.done;compares;mynum
            command = c.report
              out ! c.report
            command = c.init
              SEQ
                in ? seed;window.width;rmax;
                     stripsize;lastT;buffer.size
                out ! c.init;seed;window.width;rmax;stripsize;
                    lastT;buffer.size
            command = c.test
              SEQ
                in ? command
                out ! c.test;command
            TRUE
              SKIP
:
--------------------------------------------------------------------------
PROC mix (CHAN OF ANY mine,theirs,out)
--PROC mix acts as the control point for sending results from this
--worker or from a downstream worker depending on whether the active
--input channel is from PROC my.buffer or PROC buffer respectively.
--------------------------------------------------------------------------
  -- declarations
  #USE "raycom.tsr"   -- ray tracer command definitions
  INT command,lastT,buffer.size:
  INT x,y,address:
  INT32 seed:
  INT mynum,packets.done:
  INT64 compares:
  INT randnum,window.width,rmax,stripsize:
  [128] BYTE pixels:
  BOOL active:

  SEQ
    active := TRUE
    WHILE active
      ALT
        mine ? command
          IF
            command = c.result
              SEQ
                mine  ? x;y;[pixels FROM 0 FOR stripsize];address
                out ! c.result;x;y;
                      [pixels FROM 0 FOR stripsize];address
            command = c.init
              SKIP
            command = c.test
```

```
                SEQ
                  mine ? command
                  out ! c.test;command
              command = c.report.data
                SEQ
                  mine  ? packets.done;compares;mynum
                  out ! c.report.data;packets.done;compares;mynum
              TRUE
                SKIP

          theirs ? command
            IF
              command = c.result
                SEQ
                  theirs  ? x;y;[pixels FROM 0 FOR stripsize];address
                  out ! c.result;x;y;
                         [pixels FROM 0 FOR stripsize];address
              command = c.init
                SEQ
                  theirs ? seed;window.width;rmax;
                            stripsize;lastT;buffer.size
                  out ! c.init;seed;window.width;rmax;stripsize;
                      lastT;buffer.size
              command = c.test
                SEQ
                  theirs ? command
                  out ! c.test;command
              command = c.report
                out ! c.report
              command = c.report.data
                SEQ
                  theirs  ? packets.done;compares;mynum
                  out ! c.report.data;packets.done;compares;mynum
              TRUE
                SKIP
  :

  PAR
    my.buffer (from.ray.tr,mine)
    buffer (results.in,theirs)
    mix (mine,theirs,results.out)
:
--------------------------------------------------------------------------
PROC gen.pixel (CHAN OF ANY in,requestmore,out,VAL INT mynum)
--PROC gen.pixel processes the workpackets by generating random
--numbers until one of them falls within a specific tolerance
--(window.width) of a random number generated by PROC gen.work in the
--controller process.  The results are sent to the local PROC
--my.buffer and then to PROC mix for transmission to the controller
--either directly or via an intermediate worker.
--------------------------------------------------------------------------
  -- declarations
  #USE "raycom.tsr"   -- ray tracer command definitions
  #USE "\tdsiolib\fpmath8.tsr"
  BOOL   active,closenough:
```

```
INT32    seed:
INT    command,address:
INT    x,y:
INT    packets.done:
INT64  compares:
INT window.width:
INT randnum:
INT stripsize:
INT rnum,rmax:
INT temp:
[128]BYTE pixels:
VAL sendmore IS 0 (BYTE):

--------------------------------------------------------------------------
PROC random (INT rnum,VAL INT rmax)
--PROC random is a call to a library process which generates a
--random number.
--------------------------------------------------------------------------
  REAL32 result:
  SEQ
    RANP (result,seed)
    rnum := INT ROUND (result*(REAL32 ROUND rmax))
:

SEQ
  active := TRUE
  WHILE active
    SEQ
      in ? command
      IF
        command = c.ray
          SEQ
            in ? x;y;randnum;address
            SEQ i = 0 FOR stripsize
              SEQ
                random (rnum,255)
                pixels[i] := BYTE .num
                random (rnum,rmax)
                closenough := FALSE
                WHILE NOT closenough
                  SEQ
                    temp := rnum - randnum
                    IF
                      temp < 0
                        temp := temp TIMES (-1)
                      TRUE
                        SKIP
                    compares := compares + 1 (INT64)
                    IF
                      window.width >= temp
                        closenough := TRUE
                      TRUE
                        random(rnum,rmax)
            out ! c.result;x;y;[pixels FROM 0 FOR stripsize];address
            requestmore ! sendmore
```

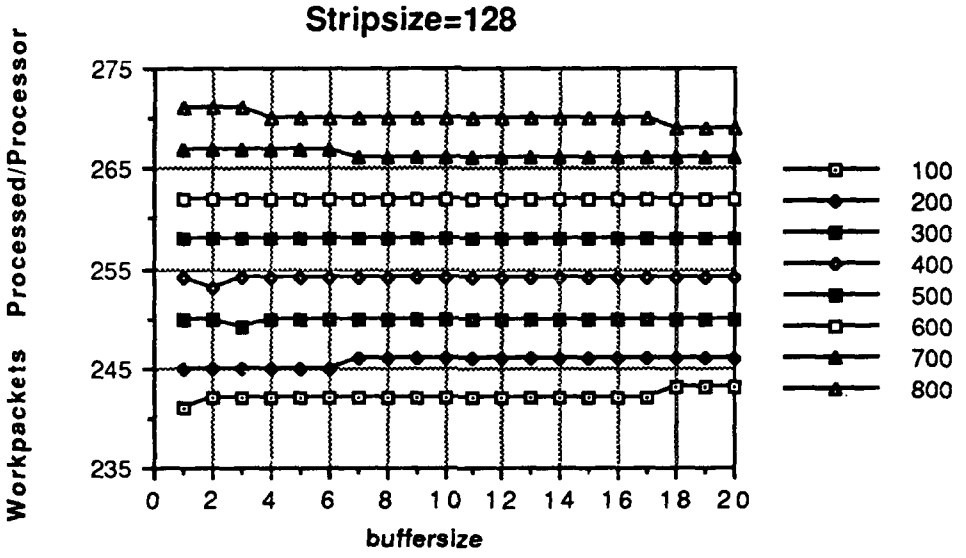
```
            command = c.init
              SEQ
                in ? seed;window.width;rmax;stripsize
                out ! c.init;stripsize
                compares := 0 (INT64)
            command = c.report.data
              SEQ
                in  ? packets.done
                out ! c.report.data;packets.done;compares;mynum
            command = c.test
              SEQ
                in ? command
                out ! c.test;command
            TRUE
              SKIP
  :

  PRI PAR
    -- high priority
    PAR
      router (work.in,work.out,to.ray.tr,requestmore,mynum)
      bypass (results.in,results.out,from.ray.tr)
    -- low priority
    gen.pixel (to.ray.tr,requestmore,from.ray.tr,mynum)
:
PROC graph(CHAN OF ANY in, out, from.mouse, to.mouse)

  #USE "\misclib\grafproc.tsr"
  #USE "\misclib\grafsymb.tsr"
  graphics(in, out, from.mouse, to.mouse)

:
PROC VT220mouse (CHAN OF ANY from.graph,to.graph,from.net,to.net)

  #USE "\misclib\term.tsr"
  #USE "\misclib\grafsymb.tsr"
  #USE "\misclib\mouse.tsr"

  mouse(from.graph,to.graph,from.net,to.net)
:

PLACED PAR


  PROCESSOR 2 T4        -- mouse/terminal process
    PLACE  mouse.to.graph AT  link0out:
    PLACE  graph.to.mouse AT  link0in :
    PLACE  from.net       AT  link2out:
    PLACE  to.net         AT  link2in :

    VT220mouse(graph.to.mouse,mouse.to.graph,from.net,to.net)

  PROCESSOR 7 T4        -- graphics process
    PLACE  to.graph       AT  link3in :
```

```
  PLACE  from.graph     AT  link3out:
  PLACE  mouse.to.graph AT  link0in :
  PLACE  graph.to.mouse AT  link0out:

  graph(to.graph,from.graph,mouse.to.graph,graph.to.mouse)

PROCESSOR 10 T8            -- controller process

  PLACE  to.graph    AT  link1out:
  PLACE  from.graph  AT  link1in :
  PLACE  work[0]     AT  link0out:
  PLACE  results[0]  AT  link0in :

  controller(to.graph,from.graph,work[0],results[0])

PLACED PAR i = 0 FOR numT8-1    -- T800 ray tracers
  -- T800 ray tracers
  PROCESSOR ((i+1)*100) T8
    PLACE  work[i]       AT  work.in[i]    :
    PLACE  work[i+1]     AT  work.out[i]   :
    PLACE  results[i]    AT  results.out[i]:
    PLACE  results[i+1]  AT  results.in[i] :

    pixel.gen(work[i],work[i+1],results[i],results[i+1],((i+1)*100))

PROCESSOR ((numT8)*100) T8
  PLACE  work[numT8-1]     AT  work.in[numT8-1]    :
  PLACE  results[numT8-1]  AT  results.out[numT8-1]:

  pixel.gen(work[numT8-1],endloopT8,results[numT8-1],
            endloopT8,((numT8)*100))
```

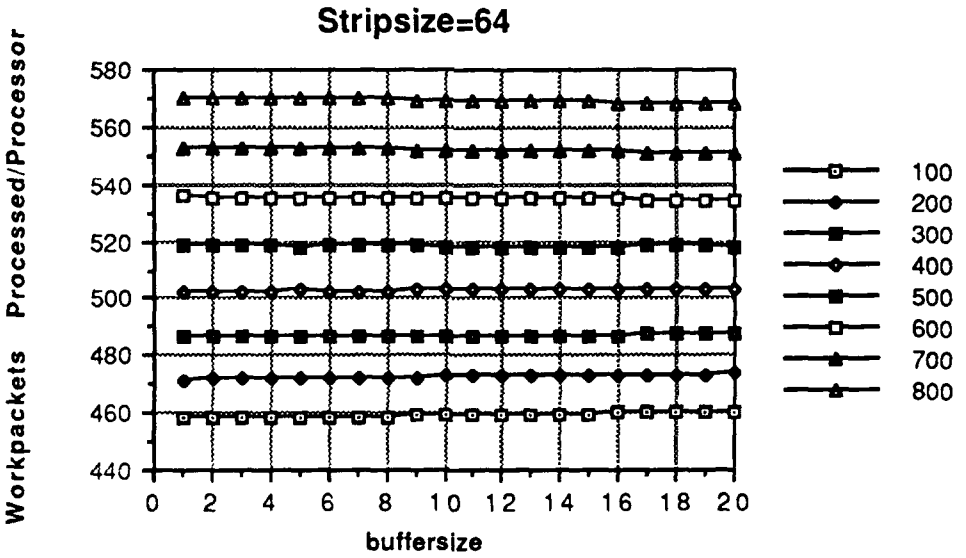
APPENDIX B

LINEAR MODEL GRAPHIC DATA (NON-ADDRESS MODE)

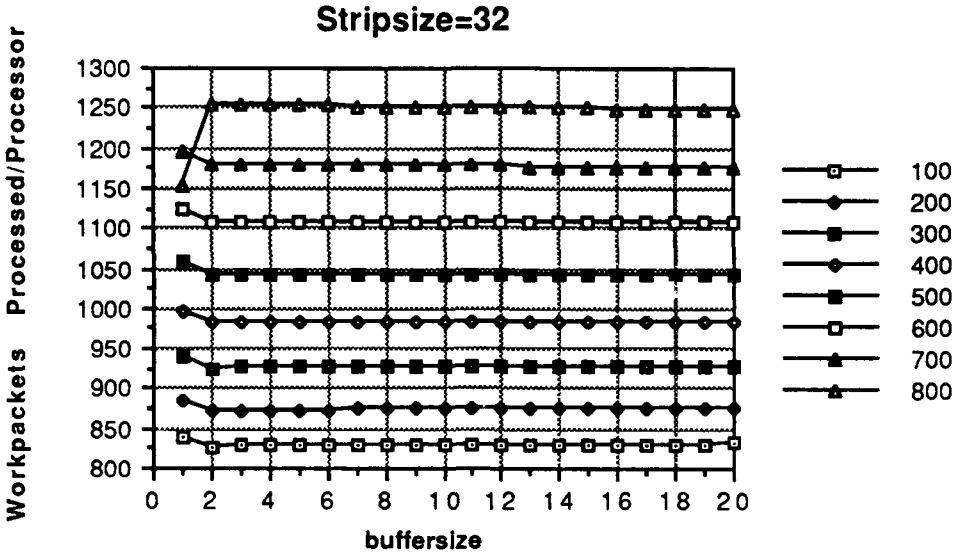
FOR WORKERS 100 THROUGH 800



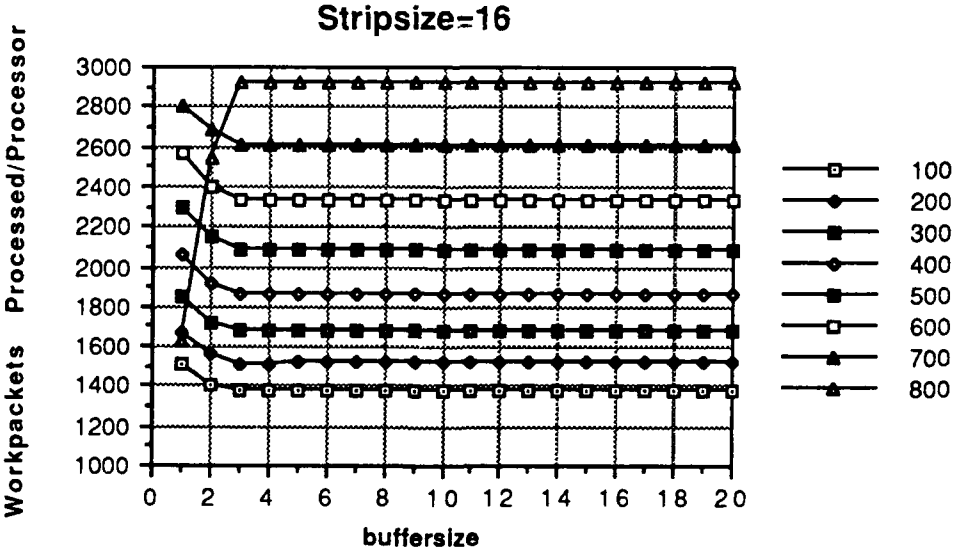
Graph B.1



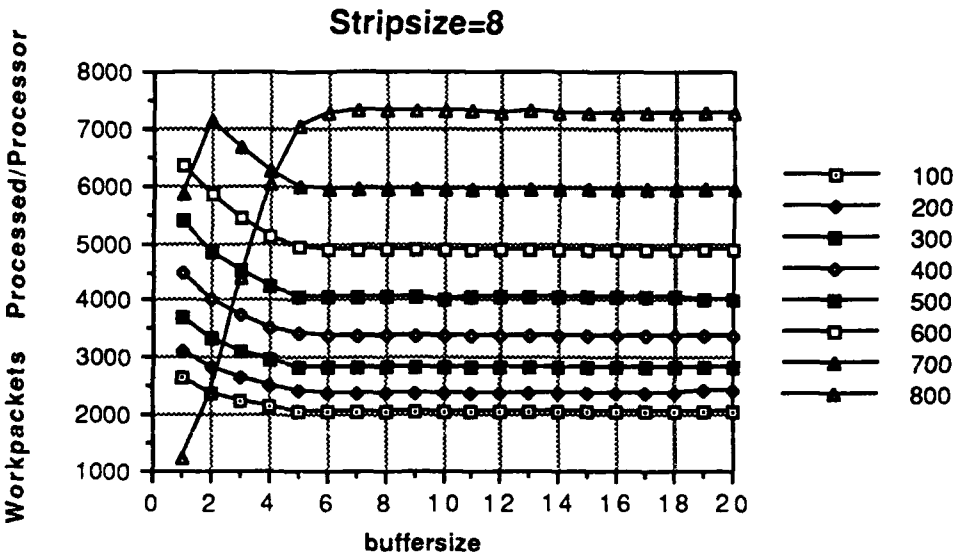
Graph B.2



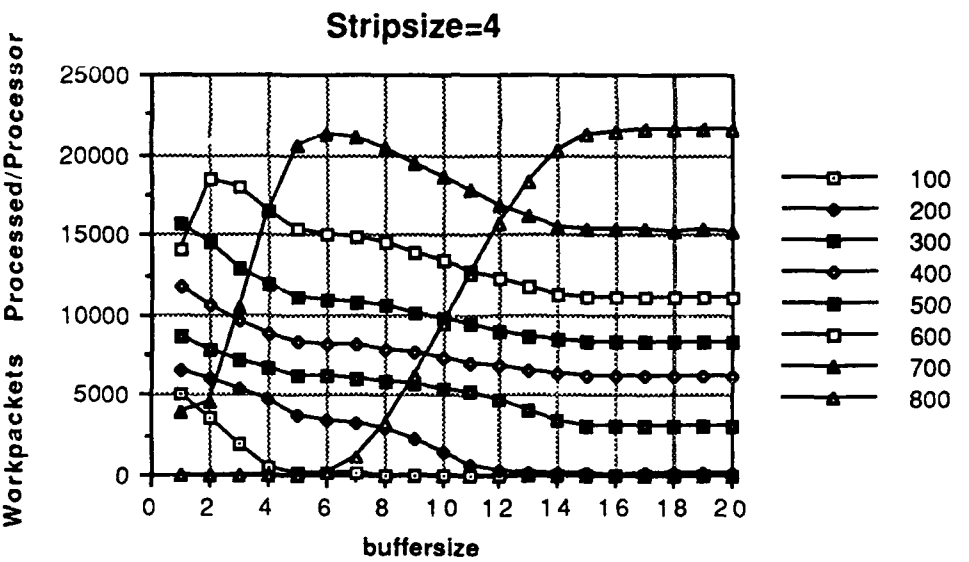
Graph B.3



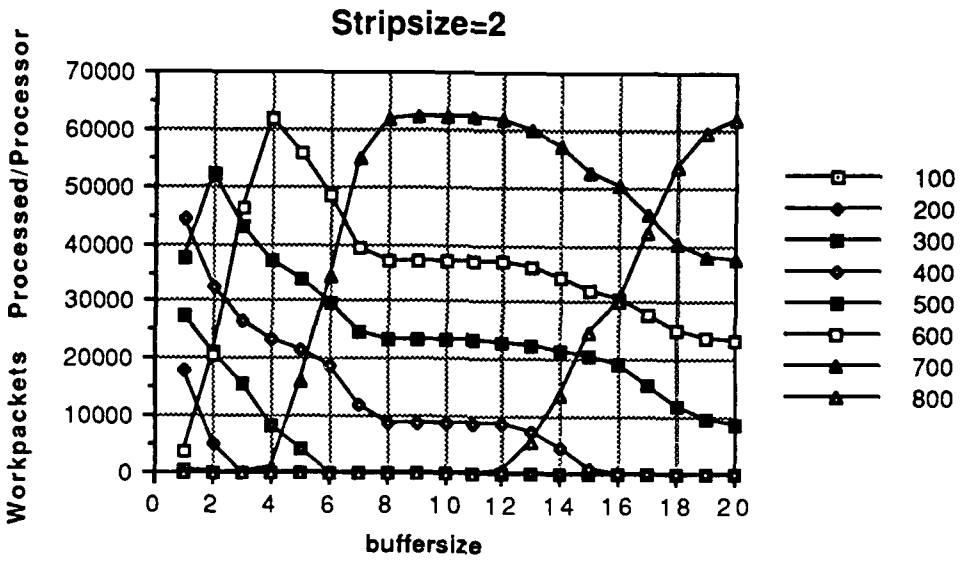
Graph B.4



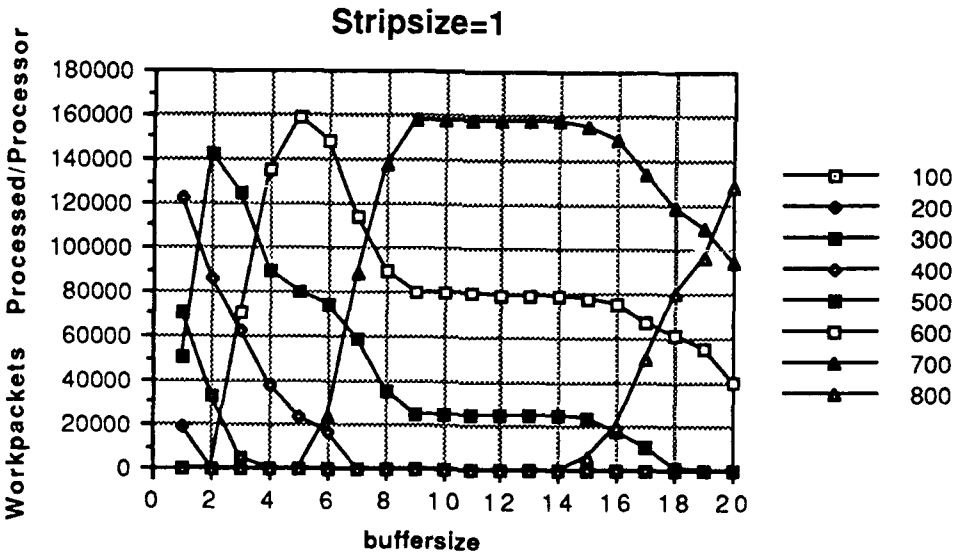
Graph B.5



Graph B.6



Graph B.7

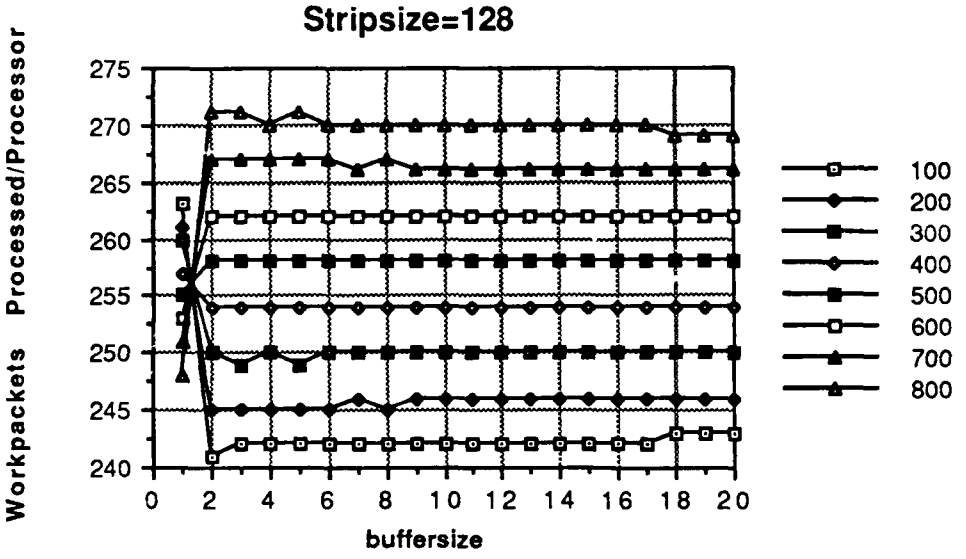


Graph B.8

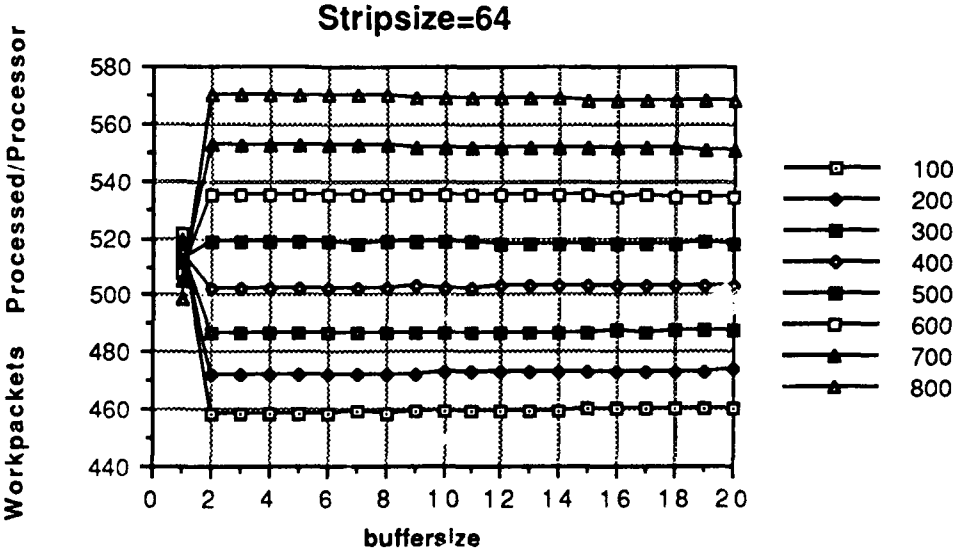
APPENDIX C

LINEAR MODEL GRAPHIC DATA (ADDRESS MODE)

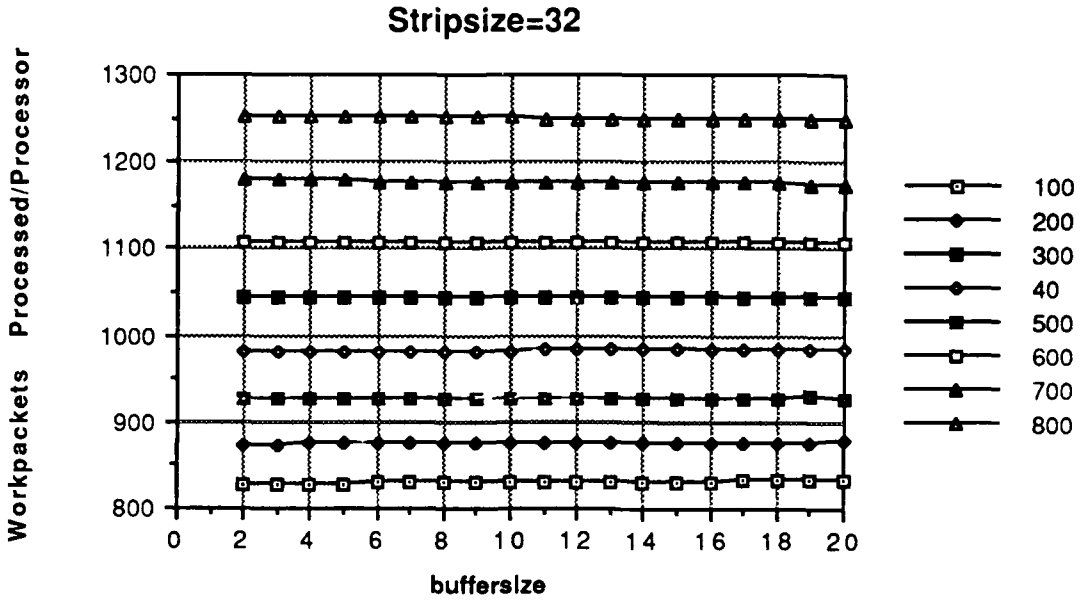
FOR WORKERS 100 THROUGH 800



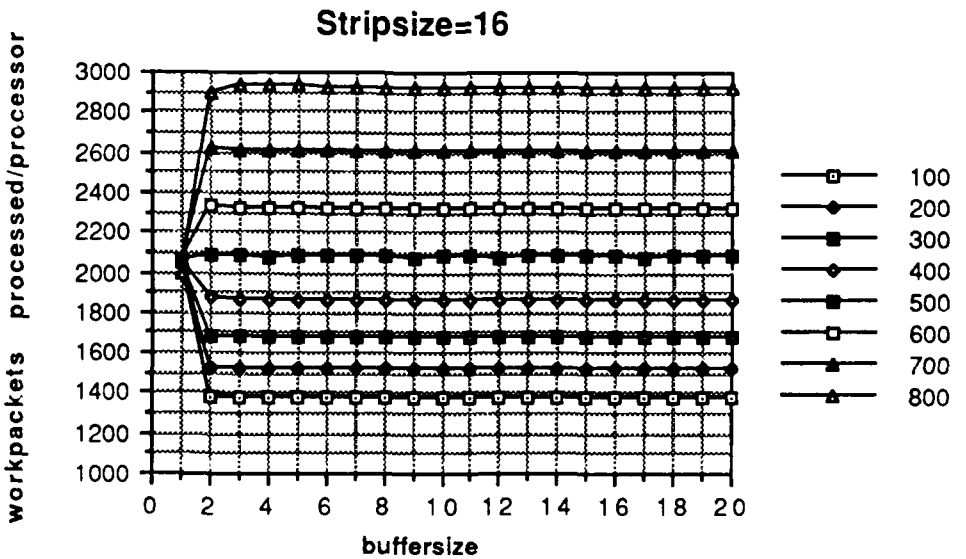
Graph C.1



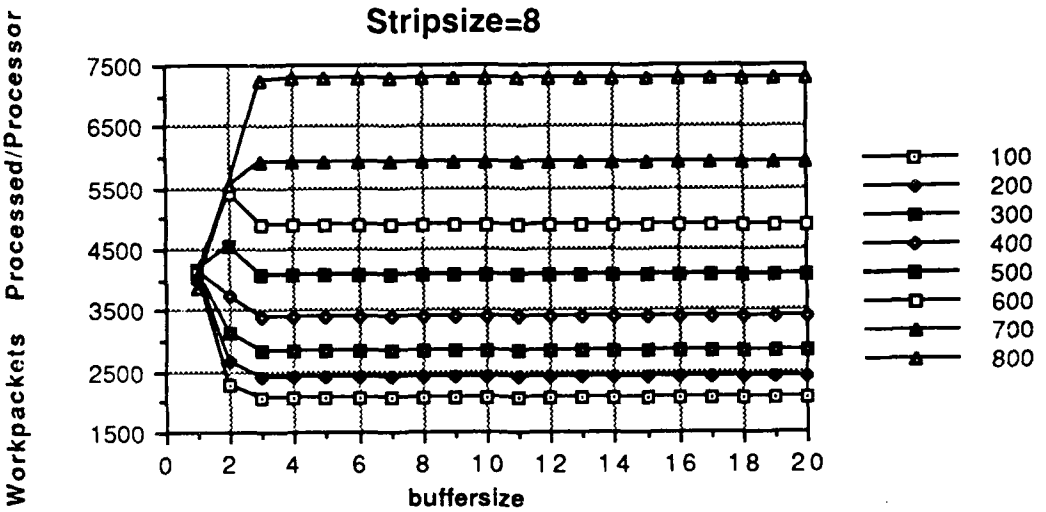
Graph C.2



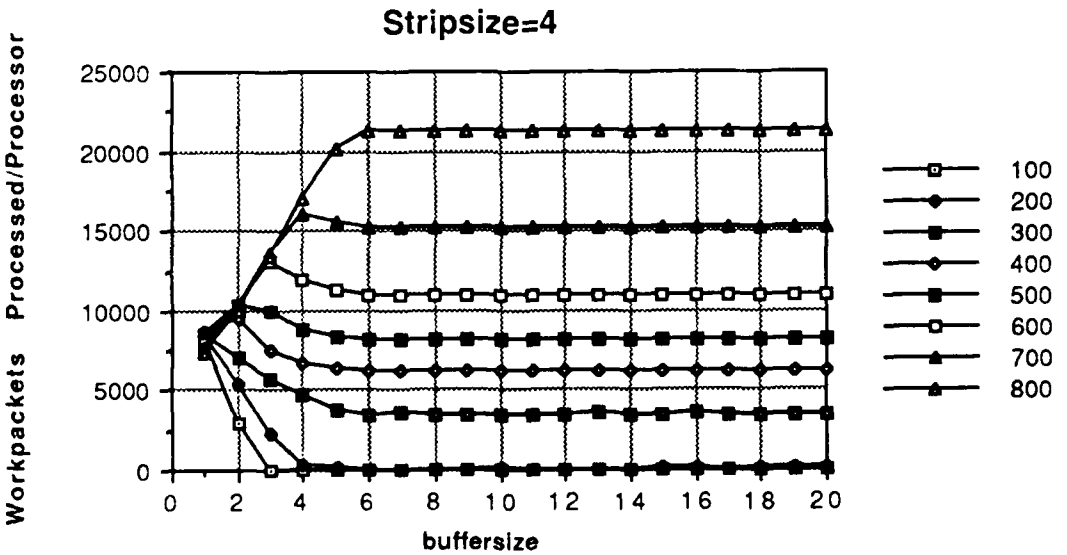
Graph C.3



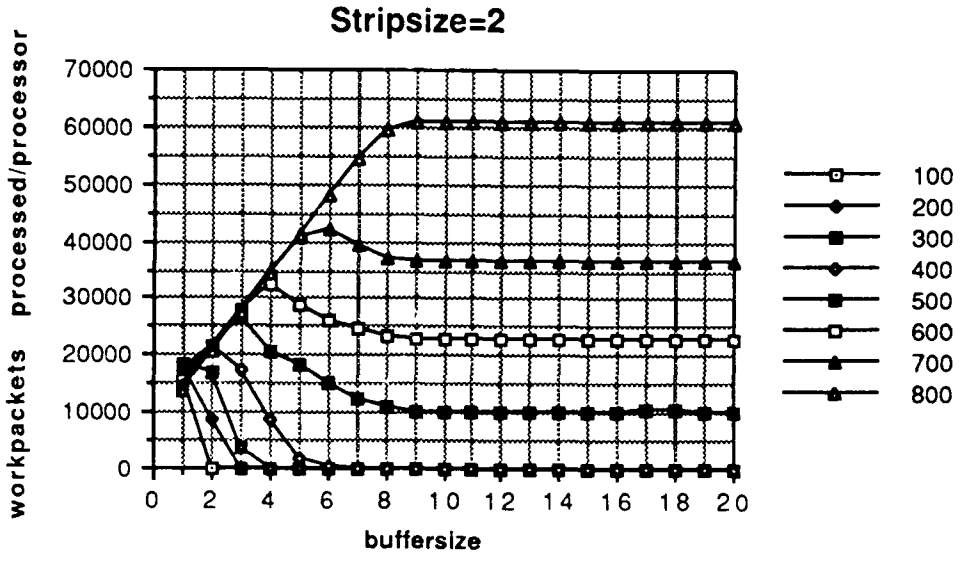
Graph C.4



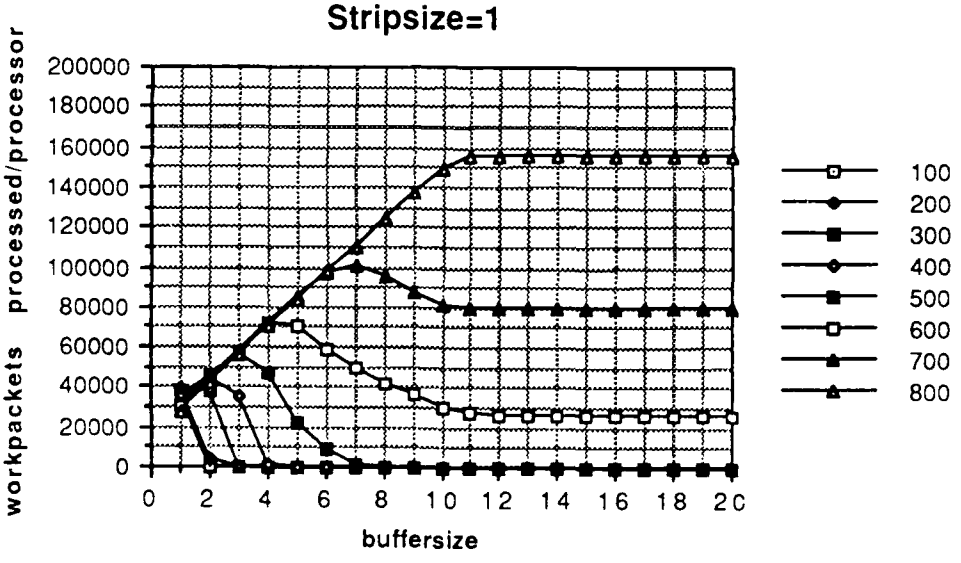
Graph C.5



Graph C.6



Graph C.7

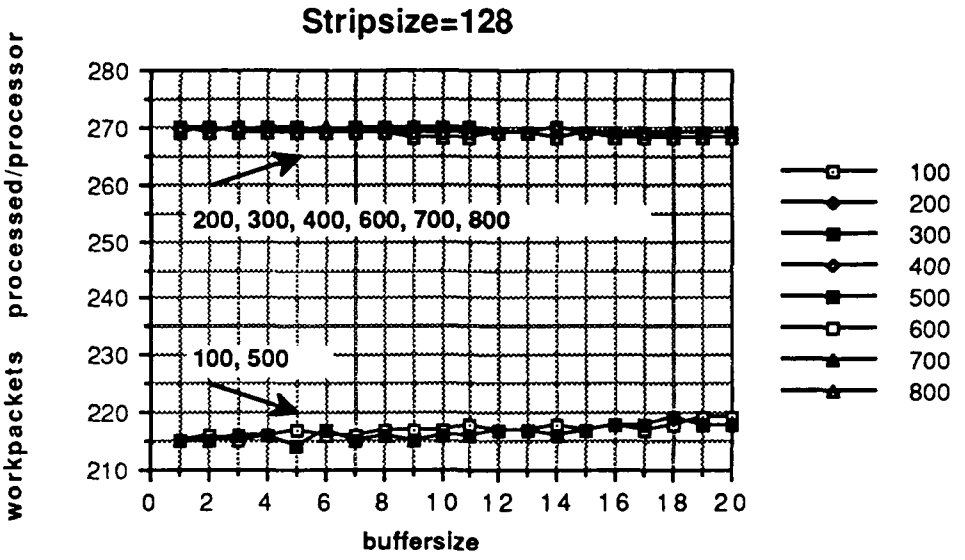


Graph C.8

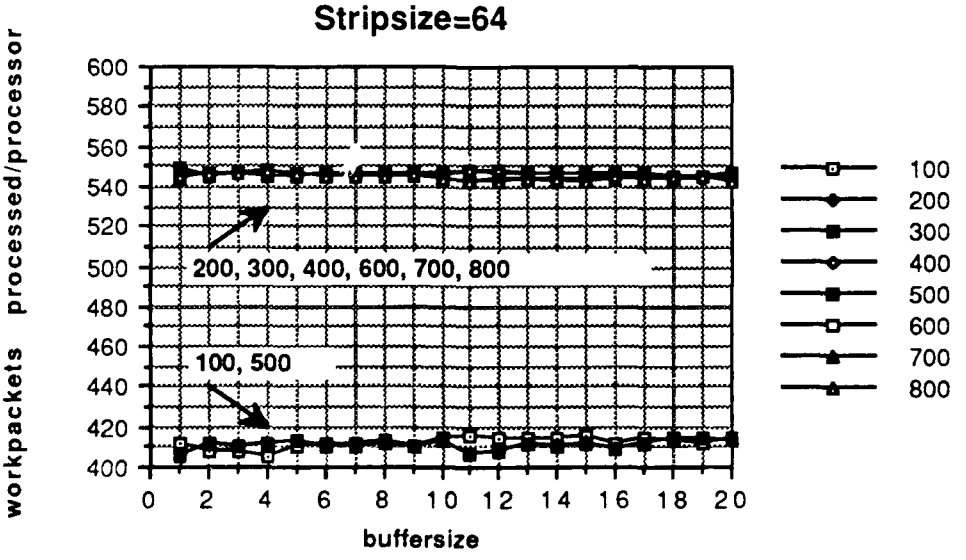
APPENDIX D

TREE MODEL GRAPHIC DATA (NON-ADDRESS MODE)

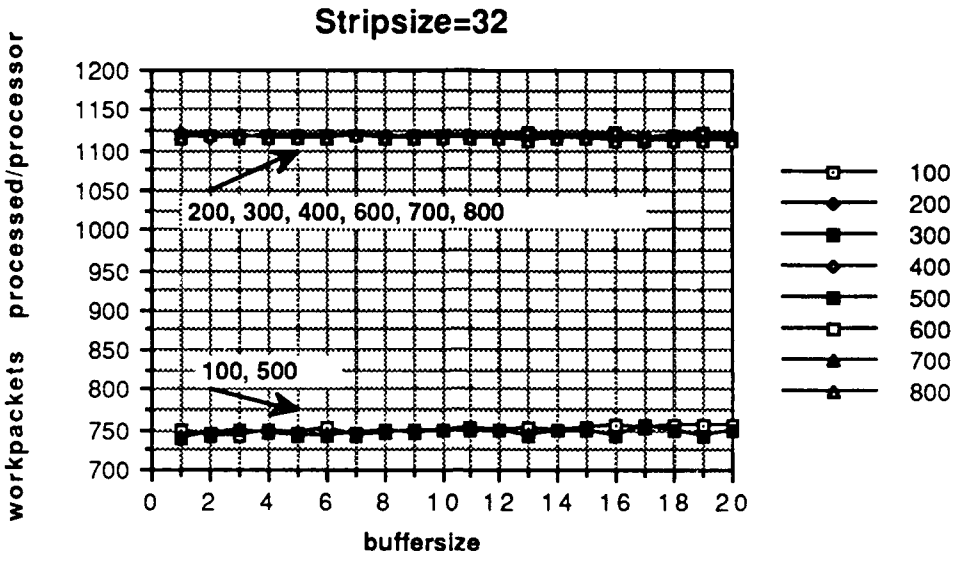
FOR WORKERS 100 THROUGH 800



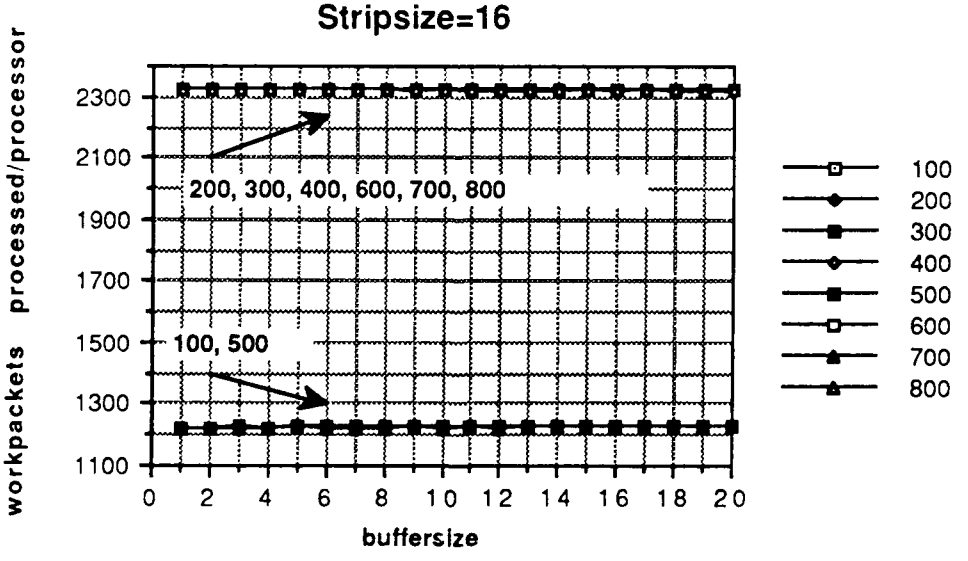
Graph D.1



Graph D.2



Graph D.3

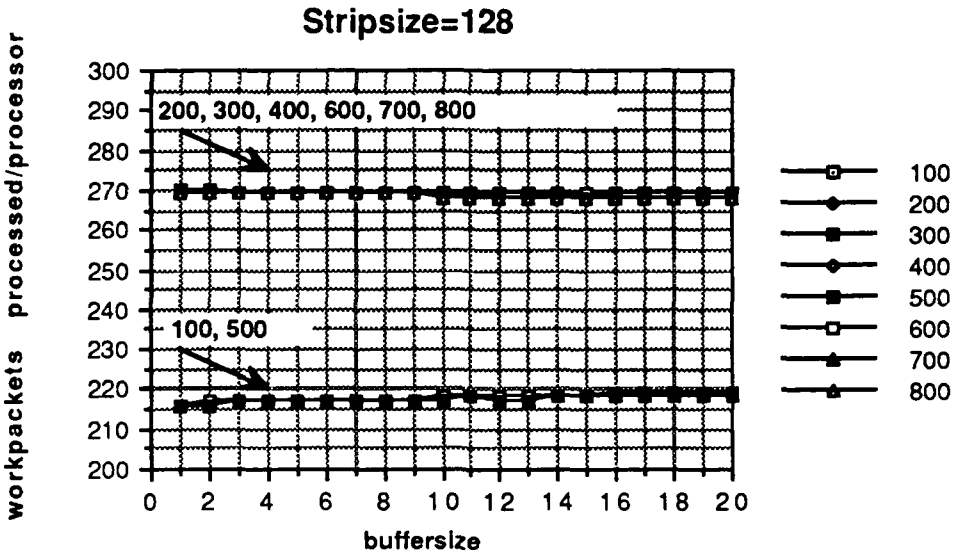


Graph D.4

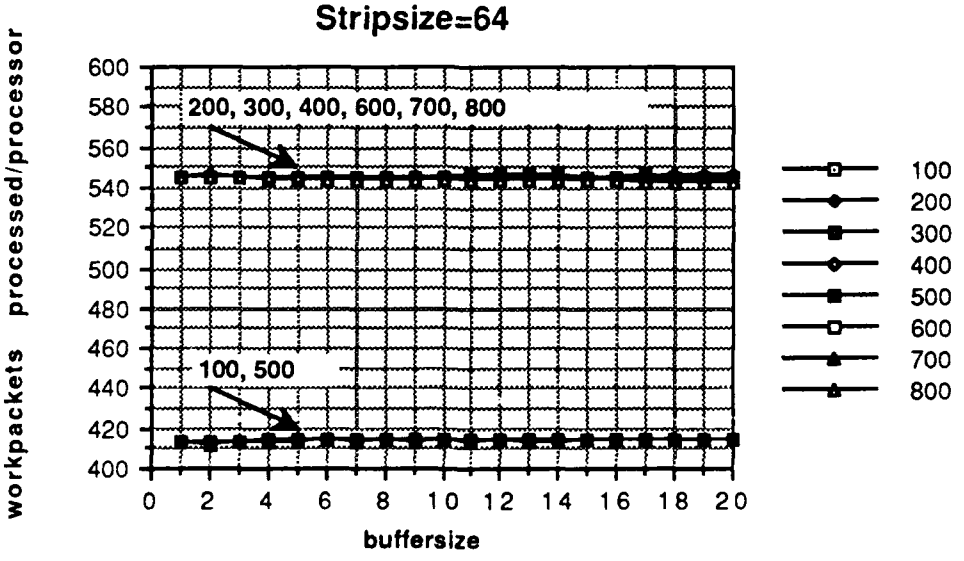
APPENDIX E

TREE MODEL GRAPHIC DATA (ADDRESS MODE)

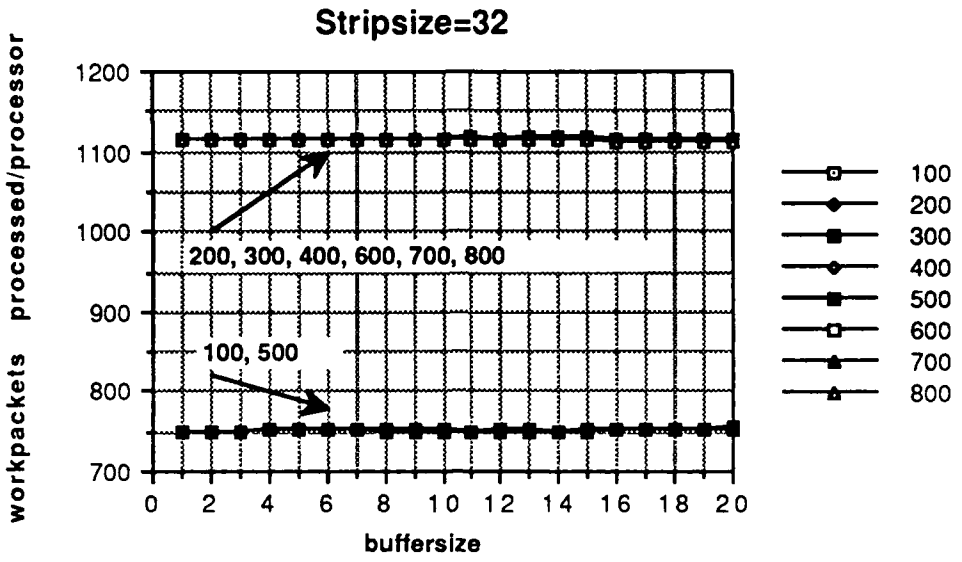
FOR WORKERS 100 THROUGH 800



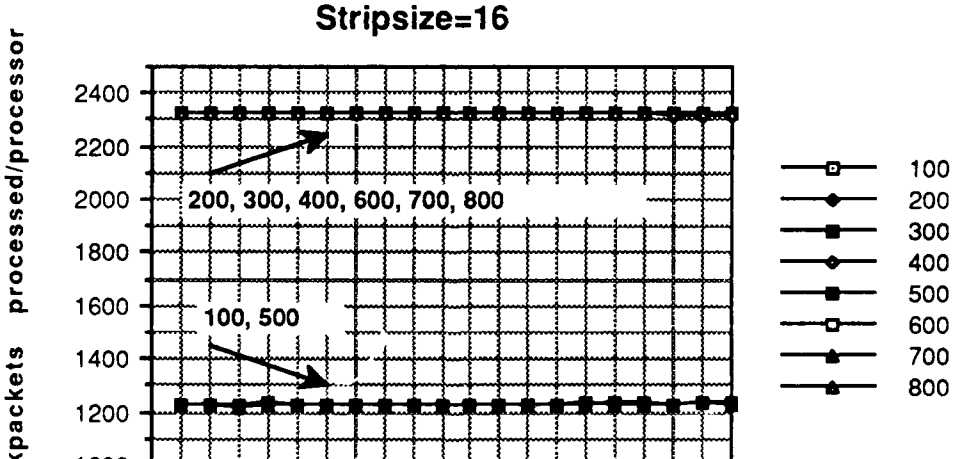
Graph E.1



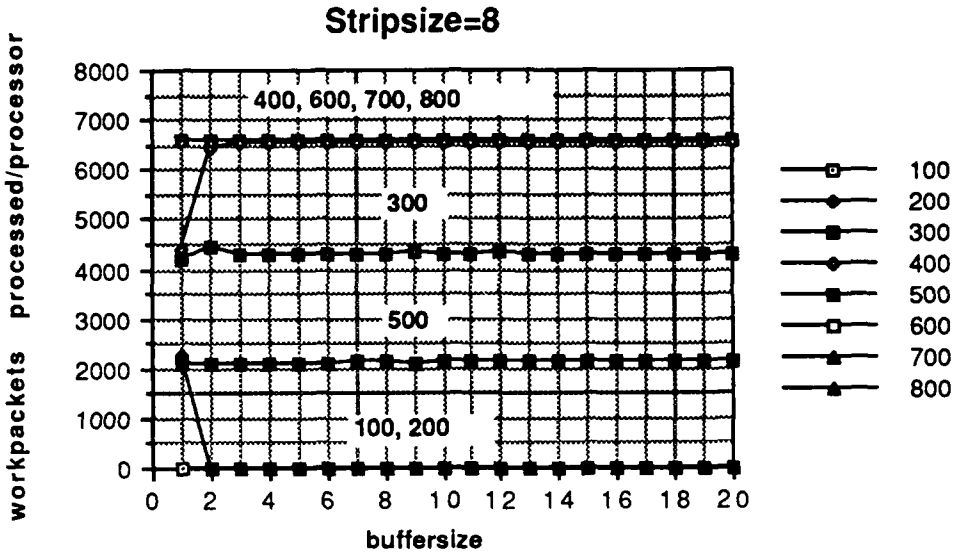
Graph E.2



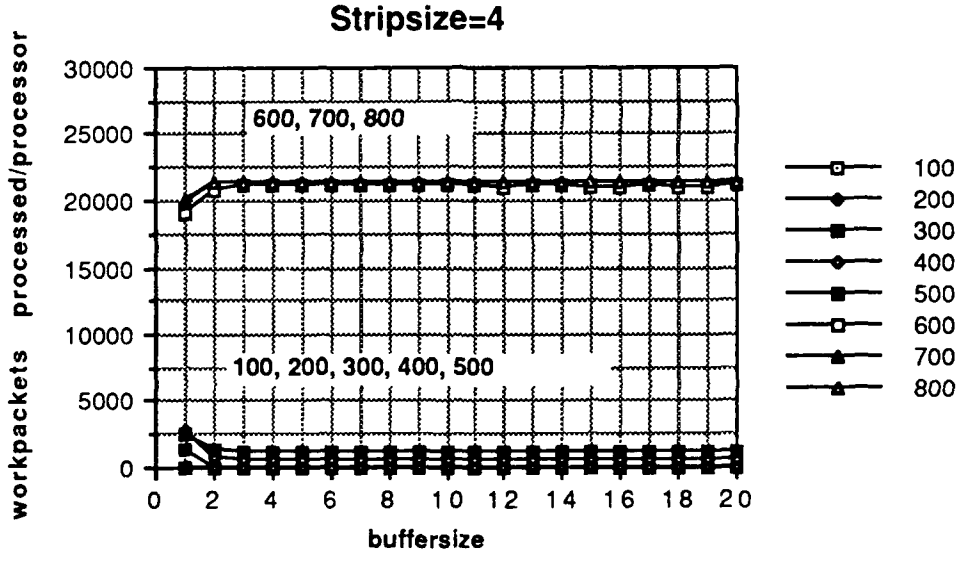
Graph E.3



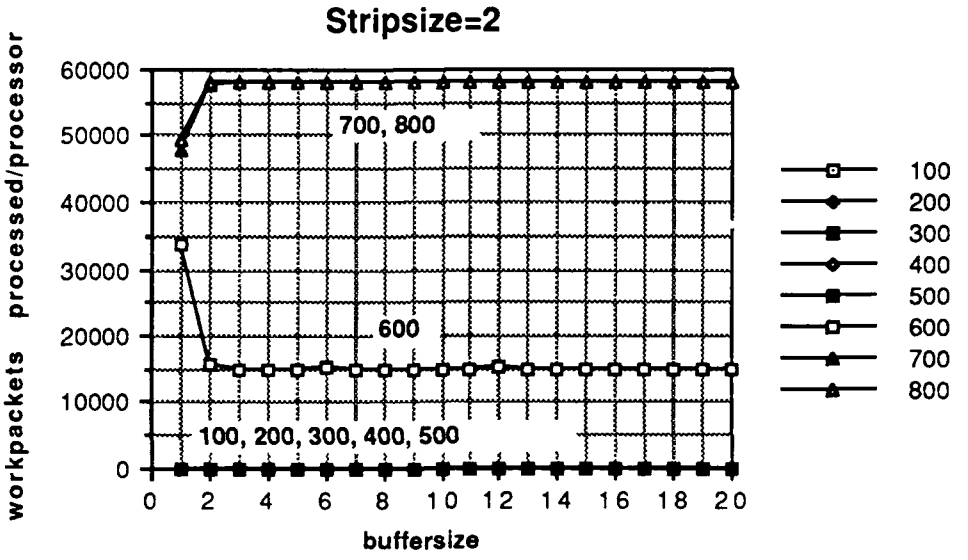
Graph E.4



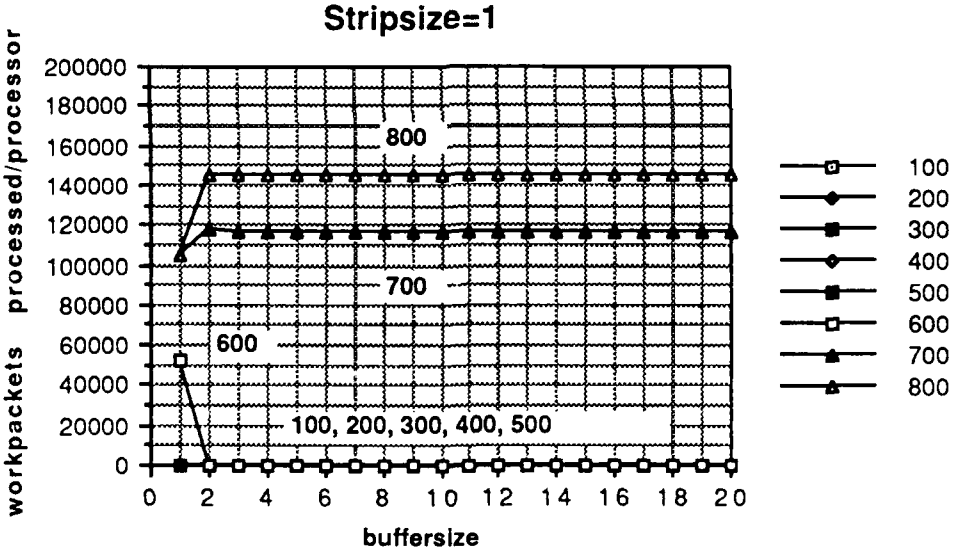
Graph E.5



Graph E.6



Graph E.7



Graph E.8

LIST OF REFERENCES

1. Stone, Harold S., *High Performance Computer Architecture*, Addison Wesley, Reading, Massachusetts, 1987.

2. *Transputer Reference Manual*, January 1987, INMOS Ltd., Bristol, United Kingdom.

3. Hoare, C. A. R., "Communicating Sequential Processes," Communications of the ACM, vol. 21, no. 8, pp. 666-677, August 1978.

4. *A Tutorial Introduction to OCCAM Including Language Definition*, March 1988, INMOS LTD., Bristol, United Kingdom.

5. Yang, C. and Johnson T. J., "A Sensitivity Analysis of the Linear Model Workfarm," accepted for presentation, SIAM Parallel Processing Conference, December 1989.

6. *Profiler: A Profiling System for Parallel Computers*, 1988, Parasoft Corporation, Mission Viejo, California.

7. *NDB: A Source Level Debugger For Parallel Computers, "A Guide to Debugging C and Fortran Programs"*, 1988, Parasoft Corporation, Mission Viejo, California.

BIBLIOGRAPHY

Bryant, Gregory R., Design, "Implementation and Evaluation of an Abstract Programming and Communications Interface for a Network of Transputers," M.S. Thesis June 1988, Naval Postgraduate School, Monterey, California.

Cloughley, William R., "Evaluation of Work Distribution Algorithms and Hardware Topologies in a Multitransputer Network," M.S. Thesis June 1988, Naval Postgraduate School, Monterey, California.

Pountain, D., "Turbocharging Mandelbrot," BYTE, September 1986, pp. 359-366.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Department Chairman, Code 62 Department of Electrical Engineering Naval Postgraduate School Monterey, California 93943	1
4.	Dr. Chyan Yang, Code 62YA Department of Electrical Engineering Naval Postgraduate School Monterey, California 93943	6
5.	Dr. Uno R. Kodres, Code 52KR Department of Computer Science Naval Postgraduate School Monterey, California 93943	2
6.	Dr. Man-Tak Shing, Code 52SH Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
7.	L/C C.N. Chih Lun Chou SMC 2460 Naval Postgraduate School Monterey, California 93943	1
8.	Mr. Marciano P. Pitargue Software Test Engineer 6607 Kaiser Drive Fremont, California 94555	1
9.	Captain Rod Scott SMC 2251 Naval Postgraduate School Monterey, California 93943	1

10.	Mr. Shawn DeKalb 10263-4 Bell Gardens Dr. Santee, California 92071	1
11.	Mr. James F. Johnson RD #1 Granville, New York 12832	1
12.	Lieutenant Timothy J. Johnson 16229 Arena Pl. Ramona, California 92065	5
13.	AEGIS Modeling Laboratory, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943	2