NASA Technical Memorandum 101297
ICOMP-88-14

Implementing Direct, Spatially Isolated Problems on Transputer Networks

Graham K. Ellis
Institute for Computational Mechanics in Propulsion
Lewis Research Center
Cleveland, Ohio

August 1988

NASA

INSTITUTE FOR COMPUTATIONAL MECHANICS IN PROPULSION
NASA LEWIS RESEARCH CENTER
ICOMP
CASE WESTERN RESERVE UNIVERSITY

IMPLEMENTING DIRECT, SPATIALLY ISOLATED PROBLEMS ON TRANSPUTER NETWORKS

Graham K. Ellis*
Institute for Computational Mechanics in Propulsion
Lewis Research Center
Cleveland, Ohio 44135

SUMMARY

Parametric studies have been performed on transputer networks of up to 40 processors to determine how to implement and maximize the performance of the solution of problems where no processor-to-processor data transfer is required for the problem solution (spatially isolated).

Two types of problems were investigated in this study. A computationally intensive problem where the solution required the transmission of 160 bytes of data through the parallel network, and a communication intensive example that required the transmission of 3 Mbytes of data through the network. This data consists of solutions being sent back to the host processor and not intermediate results for another processor to work on.

Studies were performed on both integer and floating-point transputers. The floating-point transputer features an on-chip floating-point math unit and offers approximately an order of magnitude performance increase over the integer transputer on real valued computations.

The results indicate that a minimum amount of work is required on each node per communication to achieve high network speedups (efficiencies). The floating-point processor requires approximately an order of magnitude more work per communication than the integer processor because of the floating-point unit's increased computing capability.

INTRODUCTION

With the advent of multiple-instruction, multiple-data (MIMD) stream parallel processors comes the problem of distributing the problem of interest onto a network of processors for solution. This paper discusses some techniques that can be used for implementing direct, spatially isolated problems on transputer networks.

A direct problem is one that requires a known number of iterations to solve the problem (i.e., not iterative). The term spatially isolated refers to the class of problems that can be divided such that no data are required from any other processor for the problem solution. This, however, does not preclude the necessity of distributing and collecting initial data and final answers on the transputer network.

Direct, spatially isolated problems are investigated because of their simplicity when compared with iterative, data coupled problems such as matrix inversion. By using a simple problem, the need to perform dynamic load balancing to keep all of the processors in the network busy and the extra programming required for intelligent network communication can be eliminated.

---

*Senior Research Associate (work funded under Space Act Agreement C99066G).

In general, there are two approaches to distributing a problem onto a parallel processing network. The first is to let each processor perform the same computations but on different data. The other method is to distribute the computation itself over the network so that each processor performs only part of the numerical computation. The first method will be investigated in this paper.

Two types of direct, spatially isolated problems are studied in this paper. A computationally intensive problem and a communication intensive problem. The computationally intensive problem is a definite integral that is used to compute an approximation to pi (ref. 1). The integral is evaluated using the rectangle rule. The communication intensive problem is a mapping of the complex plane used to visually search for complex roots of polynomials (private communication with Alan Palazzolo of Texas A&M). The communication limited problem occurs because of the size of the results of the computation. The computed data are quite large and consequently cause a communication bottleneck in the network.

This paper first provides an introduction to the hardware and software features of the transputer and then the programming and performance of the computationally intensive and the communication intensive problems are discussed. A working knowledge of occam is assumed.

HARDWARE

A transputer is a microprocessor containing memory, serial links that allow point-to-point connection of networks of transputers and an optional floating-point math unit on a single VLSI chip. The transputer is a Reduced Instruction-Set Computer (RISC) which gives high-performance rates for many types of computations.

The IMS T414 transputer contains 2 kbytes of 50 ns on-chip RAM, four bidirectional serial links that can operate at 5, 10, and 20 Mbits/sec data transfer rates. The 20-MHz version of the T414 is capable of approximately 10 MIPS and 100 kflops. The IMS T800 transputer contains 4 kbytes of 50 ns on-chip RAM, four bidirectional serial links at 5, 10, 20 Mbits/sec and an on-chip floating-point unit. A 20-MHz T800 is capable of approximately 10 MIPS and 1.5 Mflops (ref. 2). A block diagram of the T800 is shown in figure 1.

The transputer's links are autonomous Direct Memory Access (DMA) engines that upon initialization can transfer data without any processor intervention. The bandwidth of the microprocessor bus is such that all four serial links (both input and output) can operate at 20 Mbits concurrently.

The transputer system used in this study consists of a transputer host/development system that runs on a PC AT compatible. The development board contains a 15-MHz T414 processor with 2 Mbytes RAM. The transputer network consists of 40 20-MHz transputers with 256 kbytes DRAM per processor and a transputer based medium performance graphics board. The graphics board supports a resolution of 512 by 512 pixels and 256 simultaneous colors.

Simulations using either 40 T414 or 40 T800 transputers were performed to determine the advantage of the on-chip floating-point math unit.

The link speed for all the tests was set at 10 Mbits/sec. The network was wired as a 10 processor by 4 processor torus. This architecture is convenient because many other architectures of interest can be mapped onto a torus. Examples of other architectures that map onto a 10- by 4-torus are a two-dimensional mesh, ring, pipeline, and hypercubes of order 1, 2, or 3.

SOFTWARE

All of the software for the performance tests in this paper were written in occam. Occam was developed to easily implement communication and concurrency (ref. 3). Occam can be used to describe the structure of a network or system in terms of point-to-point communication channels. It is also used to program the individual processors in the network.

The software was developed using the INMOS Transputer Development System (TDS). The TDS is an integrated package consisting of an editor, occam compiler, linker, network configurer, and other development tools (ref. 4). The version used for the benchmarks was occam TDS, BETA2, D700C.

Occam allows a system to be described in terms of a collection of concurrent processes (ref. 5). A process performs a sequence of actions and terminates. Concurrent processes can only communicate with each other and with peripheral devices through point-to-point channels. Occam programs are built from three primitive processes (ref. 6).

Assignment	
v := e	assign expression e to variable v
Output	
c ! e	output expression e through channel c
Input	
c ? v	input from channel c to variable v

Constructors are used to combine primitive processes into larger processes. The sequential constructor, SEQ, causes its components to be executed one after another. A SEQ process terminates when the last component under the SEQ terminates. The parallel construct, PAR, causes its components to be executed concurrently. If a PAR is specified on a single processor, the processes are time-sliced according to a round-robin scheduler built into the transputer hardware. A PAR process terminates only after all of the components under the PAR have terminated. The alternative construct, ALT, chooses one component process for execution. If more than one component process is enabled to be selected, only one will be selected. The selection of the process is arbitrary. An ALT process terminates when the selected component terminates.

The transputer supports two priority levels for its operations. All parallel processes running at a low priority get time sliced according to the on-chip hardware scheduler built into the transputer. A low-priority time slice is 5120 cycles long (approximately 1 ms) (ref. 7). A high-priority

process will run without interruption until it finishes or has to wait for channel communication. The PRI PAR statement is used to prioritize a process. Only two process can appear under a PRI PAR statement. The first process appearing after the PRI PAR statement is run at high priority. To run more than one process at high priority, the following syntax can be used:

```
PRI PAR
  PAR
     a()
     b()
     c()
  d()
```

where a(), b(), c() and d() are occam procedures. Procedures a(), b() and c() running under the PAR construct are considered to be a single process.

The code fragment listed above means processes a(), b(), and c() will all run at high-priority until they deschedule for communication or they terminate. Each of the high-priority processes will run preemptively until they deschedule. High-priority processes do not get time-sliced. The exact ordering of the processes appearing in a PRI PAR is the last lexically appearing process will schedule first, and then the others will be queued up as they appear top-to-bottom. In the example listed above, the high-priority process list will appear as:

```
c()
a()
b()
```

When none of the processes a(), b(), and c() can proceed, process d() gets scheduled. If any of the processes in the high-priority process list get scheduled, it will interrupt the process d().

Any programs written using the PAR or PRI PAR constructs should not depend on the order that the processes are queued in process list for proper operation; however, knowing how the occam compiler generates the code can be helpful when maximum performance is required.

Channel communication is self-synchronizing. Communication occurs when one process outputs to a channel and another process inputs from it. If either the sender or receiver is not ready, the other process waits until both processes are ready to continue. The programmer does not have to explicitly specify the synchronization, it is performed automatically just by using the input and output primitives presented previously. There is no implicit buffering of channel communications. Asynchronous channel communication is not directly supported.

COMPUTATION INTENSIVE EXAMPLE

The computationally intensive problem to be solved on a transputer network is a small sized problem with very little data transfer through the network. The problem is the Pi Program (ref. 1). It computes an approximation to pi by using the rectangle rule to approximate the following definite integral:

$$\int_0^1 f(x)\,dx \tag{1}$$

where

$$f(x) = \frac{4}{1 + x*x} \tag{2}$$

and the rectangle rule states:

$$Rn(f) = h[f(xi)] \tag{3}$$

where

$$h = \frac{1}{n} \tag{4}$$

$$xi = \left(i - \frac{1}{2}\right)h \tag{5}$$

In order to implement the pi program on a network of transputers, first a suitable architecture must be chosen. Unlike many multiprocessors whose architecture is fixed, transputers can be wired in any configuration supported with four connections per processor.

Since the programmer must develop all of the communication routing algorithms for the network, it is currently more convenient to implement a simple, regular architecture and use simple communication procedures instead of a more complicated scheme. Additionally, the DMA link engines allow data to be piped through a network of processors with little performance penalty.

The problem will be implemented on a pipeline of transputers. A pipeline is used because it is very easy to implement the required communication buffers for the network data transfer. Simulations will be run for 1, 8, 16, 32, and 40 processors. Both T414 and T800 versions of the transputer will be tested.

As a baseline, the pi program was implemented on a single transputer (the development board) using only SEQ processes. This version gives a datum from which the network computations can be compared. The listing of the single processor sequential version of the pi program is given in appendix A.

The next step was to simulate the desired network of transputers on a single transputer using the PAR construct. One of the advantages of occam is the ability to simulate the network configuration on a single processor. The discussion for the simulated parallel network and the actual parallel network are combined since they are conceptually identical. The listing of the single processor parallel version of the pi program is presented in appendix B and the network version is listed in appendix C.

As previously mentioned, a pipeline of processors (or processes for the single processor case) is used because of the simple communication protocol required. The current generation of software tools for transputers is such that the programmer must explicitly define and program the processor-to-processor data routing procedures. The pipeline and communication buffers are shown in figure 2.

The single processor sequential implementation of the rectangular integration is quite simple. The occam code fragment for the rectangle rule computation is shown below.

```
SEQ i = 0 FOR number.intervals
  SEQ
    xi := ((REAL32 TRUNC i) - 0.5 (REAL32)) * delta.x
    sum := sum + (delta.x * (4.0 (REAL32) / (1.0 (REAL32) + (xi * xi))))
```

where

number.intervals	the total number of intervals to use for the integration
delta.x	the width of each interval in the integration
xi	temporary storage
sum	the value of the integral
i	loop counter

Note the strict data typing occam requires. This strict typing insures the correctness of any occam expression.

The obvious method for distributing the integration onto a network of N processors is to divide the interval [0, 1] into N equal segments and let each processor work on a subset of the interval. The real work for implementing the parallel solution of the pi problem is in writing the communication routines for the network.

Since the number of processors is known at compile time, each processor can be assigned a unique number that can be used to determine the start and end values for the interval on each processor. The computation required to compute a local interval [u, v] is as follows:

$$u = n\left(\frac{c}{N}\right) \tag{6}$$

$$v = u + \frac{c}{N} \tag{7}$$

where

n	processor number (numbers start at 0)
c	total number of intervals on [0, 1]
N	total number of processors in network

Note that the number c/N is a constant. Because of this, the host computer can compute it once and send it to every processor. The value c/N is really the number of local intervals to be computed on each processor.

The formulation for the rectangle rule requires the multiplier value $h = 1/c$ for the proper function evaluation. Therefore, the only values

required to be output to the network to start the simulation is to output the value for h, and the value for c/N.

INPUT BUFFER PROCESS

To distribute the work onto the network, some sort of input buffer routine must be written. Since the data needs to propagate down the pipe, the obvious scheme is to read the data from the host and pass it to the local computation process and also to the next processor (process) in the network. The data can be sent from the input buffer to the local compute process and the next processor in parallel. This also increases the performance of the network since the serial links can operate autonomously without processor intervention once the initial communication is set up. The channel set-up time for a communication is approximately 1 μsec (20 machine cycles) (ref. 8). The code fragment showing the input buffer routine is given below:

```
SEQ
  in ? delta.x; local.intervals
  PAR
    to.local.compute ! delta.x; local.intervals
    to.next.processor ! delta.x; local.intervals
```

where

delta.x	the width of each interval in the integration
local.intervals	the number of intervals to be computed on this processor
in	the input channel
to.local.compute	the channel to the computing process on this node
to.next.processor	the channel to the input buffer on the adjacent processor

COMPUTE PROCESS

The compute process is similar to the sequential program version; the local start and stop points [u, v] are used instead of the whole interval [0, 1]. The whole process consists of reading in delta.x and the number of local intervals from the input buffer process, and then performing the required computations. The partial sum is then sent to the output buffer process. The code fragment showing the computation process is given below:

```
SEQ
  from.input.process ? delta.x; local.intervals
  sum := 0.0 (REAL64)
  SEQ i = (processor.number * local.intervals) FOR local.intervals
    SEQ
      xi := ((REAL64 TRUNC i ) - 0.5 (REAL64)) * delta.x
      sum := sum + (delta.x * (4.0 (REAL64)) / (1.0 (REAL64) + (xi * xi))
                to.output.process ! sum
```

where

delta.x	the width of each interval in the integration
local.intervals	the number of intervals to be computed on this processor
from.input.process	the input channel from this node's input buffer
to.output.process	the channel to the output buffer process on this node
i	loop counter

OUTPUT BUFFER PROCESS

Once the local data has been computed, the partial sum needs to be sent back to the host so it can be combined with the results from the other processors. The output buffer needs to be able to read data from both the local compute process and the adjacent processors output node. Since the order of the results appear on the network is unknown, the occam ALT construct is used. The code fragment showing the output buffer routine is shown below:

```
WHILE TRUE
  ALT
     from.local.compute ? local.sum
       link.out ! local.sum

     from.adjacent.processor ? sum
       link.out ! sum
```

where

local.sum	temporary storage for the partial sum from the local compute process
sum	temporary storage for the partial sum from the adjacent node's output buffer process
from.local.compute	the channel from the local compute node
from.adjacent.processor	the channel from the adjacent node's output buffer process

HOST PROCESS

The host process starts the simulation by sending out the desired delta x and number of local intervals. The host process then waits for the results from the network. Since the number of processors in the network is known, the number of data packets received is recorded and when all packets have been received from the network, the program terminates. The occam code fragment for the host process is listed below:

```
      out ! h; cn
      replys := 0
      total := 0.0 (REAL64)
      WHILE replys < num.processors
        SEQ
        in ? partial.sum
        total := total + partial.sum
        replys := replys + 1
```

where

out	the output channel to the first processor in the pipeline
h	(1/number of total intervals) on [0, 1]
cn	c/N, the number of local intervals on each processor
replys	the number of replys received from the network
num.processors	number of processors in network
total	the approximate value for pi
partial.sum	a single processor's contribution to total
in	the input channel from the first node in the pipeline

PERFORMANCE

The pi program was evaluated on 1, 8, 16, 32, and 40 transputers. For each case, the number of total intervals for the integration was varied from 10^3 to 10^7 in powers of 10. The first performance tests were to determine the optimum priorities for computation and communication.

The computation times were obtained using the occam TIMER statement to read the on-chip timer. The low-priority timer has a resolution of 64 μsec per tick.

The recommended method of programming transputer networks is to assign a high priority to communication and low priority to computation (ref. 9). To verify this, three different process priority configurations were tested. Each case uses three processes that run in parallel on each node. The three cases are as follows:

Prioritized communication:

```
PRI PAR
  PAR
    input.buffer()
    output.buffer()
  compute()
```

Prioritized computation:

```
PRI PAR
  compute()
  PAR
    input.buffer()
    output.buffer()
```

All processes low-priority:

```
PAR
  input.buffer()
  output.buffer()
  compute()
```

The results for the communication tests for a T800 floating-point transputer network are shown in figures 3 to 5. The optimum case is to assign priority to communication. The reason that prioritizing communication increases network performance is that more processors can be kept busy. By interrupting processing to set up a data transfer, other processors in the network receive data to work on rather than waiting until the adjacent processor finishes its computations. Another reason that it helps to prioritize communication is that the transputer links, once initialized, can transfer data without processor intervention. By prioritizing the data transfer, concurrent computation, and communication can occur on a single transputer.

Given that priority should be given to communication, the pi program benchmark was run on both T414 and T800 networks. Both 32-bit and 64-bit math version were tested. The results from these tests for the T800 floating-point processors are shown in figures 6 and 7. A comparison of the floating-point performance of the T414 and T800 shows that the T800 has approximately an order of magnitude increase in performance over the T414 for 32-bit computations. There is approximately a 40 times speedup on the T800 over the T414 on 64-bit floating-point computations. This can be reconciled by the fact that the T800 has a 64-bit floating point unit built into the hardware and the T414 must build 64-bit numbers out of 32-bit operands which typically takes four times as many operations as 32-bit computations.

Note, however, that for a small number of total intervals for the integral computation that the network performance is faster for 8 processors when compared to 40 processors. The extra communication time for distributing a small workload over 40 processors causes a decrease in performance over the 8 processor case since less communication time is spent distributing the work and each processor has more data to work on.

Normally the speedup of a network is defined as:

$$\text{Speedup} = \frac{\text{Solution time for 1 processor}}{\text{Solution time for N processors}} = \frac{t1}{tn} \tag{8}$$

or often the speedup can be normalized to express efficiency. One hundred percent efficiency means there is no overhead for communication on the network.

$$\text{efficiency (\%)} = \frac{t1(100)}{tn(N)} \tag{9}$$

Since the development board being used is a 15-MHz T414 it is not meaningful to compare the development board results with the 20-MHz T414 and T800 network simulations. The first method of generating single node timings for speedup computations was to use the development board and a single T800. This should not be as fast as a single T800 running the problem because of network overhead.

In order to get results for a single processor, a B004 development board was modified to use a 20-MHz T800 floating-point transputer. Also, the modified B004 development board changed the number of wait states on the external memory from five-cycle at a 15-MHz clock to three-cycle at a 20-MHz clock which equates to going from a 330-ns memory cycle to a 150-ns memory cycle. This modification still makes the comparison to the network nodes difficult since they use four-cycle memory (200-ns cycle). Another complicating factor is that the transputer loader used for these tests does not load the on-chip RAM on the development board so the 50-ns on-chip memory cannot be used and memory access times are not the same as the processors on the network. The times for the development board T800 could be expected to be at most 25 percent faster than would be expected using a four-cycle external memory.

The results for the 20-MHz T800 development board, a 15-MHz T414 development board, a single 20-MHz T800 network node (using a 15-MHz T414 as a host) and a PC/AT 80286/80287 at 8 MHz are shown in figure 8. The single-node network T800 was about 33 percent faster than the T800 on the development node. This is probably due to the loader not taking advantage of the on-chip memory on the development board. The use of the on-chip memory should make the T800 on the development board run faster since there is no network overhead for a single processor. The T414 shows performance approximately an order of magnitude slower than the T800. The 80286/80287 chip set at 8 MHz is approximately 50 percent slower than the T414 at 15 MHz.

Speedup and efficiency computations using the single T800 on the development board as $t1$ and the network times as tn are shown in figures 9 and 10. Figure 9 shows the speedup of the network for 8 to 40 processors. for less than 10000 intervals, the 8 processor network is the fastest. Figure 10 shows the same data as figure 9 except the speedup has been normalized by the number of processors to express efficiency. The difficulty in using figure 10 is that execution time must be computed from the plot. It is difficult to tell how much faster, if any, the 8 processor case is when compared to the 40 processor case for 1000 intervals.

Instead of just measuring $t1/tn$ for speedups, another method of quantifying speedup is splitting the solution time into communication and processing components. This can be used to determine the maximum possible network performance as a function of the ratio of communication time to processing time. This equation assumes there is no sequential (serial) part of the program to slow down the network. The problem is perfectly parallelizable. The speedup equation is written in terms of how long does it take to implement a sequential problem on a parallel network. The following equation is presented in reference 10.

$$\text{Speedup} = \frac{Tp}{Tc + \left(\frac{Tp}{N}\right)} \qquad (10)$$

where

Tp processing time for one processor

Tc communication time for concurrent solution

N the number of processors in the network

The speedup equation (10) can be rewritten as

$$\text{Speedup} = \frac{N}{N\left(\frac{Tc}{Tp}\right) + 1} \tag{11}$$

This formulation of the speedup equation, however, does not compare the parallel communication time, Tc, with the parallel processing time. The processing time, Tp, is for a single processor while Tc is the communication time over the distributed network.

Another method of presenting the speedup equation is to base it on the time to implement a parallel problem on a sequential network (ref. 11). In this case, the following equation can be used for speedup:

$$\text{Speedup} = \frac{N(Tp')}{Tc' + Tp'} \tag{12}$$

where

Tc' communication time for network solution

Tp' processing time per node on the network solution

N the number of processors in the network

Rewriting equation (12) yields:

$$\text{Speedup} = \frac{N}{\left(\frac{Tc'}{Tp'}\right) + 1} \tag{13}$$

This equation presents the proper parallel communication time divided by parallel processing time and gives the correct speedup ratios. The serial processing time is this case is also assumed to be zero.

The ideal case is zero network communication time. In this case, the speedup is merely the number of processors. For maximum performance, the ratio (Tc/Tp) should be minimized. A plot of speedup as a function of the number of processors is shown in figure 11.

Obviously, the Tc/Tp ratio can be minimized by either minimizing Tc by using a processor with high-speed data transfers from node-to-node or maximizing Tp by using slow processors or a large number of operations per communication. Since the T800 performs floating-point arithmetic with exceptional

speed, considerable work needs to be allocated to each node between communications for maximum performance. A reasonable goal seems to be a Tc/Tp ratio of at most 0.001, (i.e., 1000(Tc) = Tp).

NETWORK COMMUNICATION LIMITED PROBLEM

The problem tested on the network to cause a communication bottleneck was a simple mapping of a region of the complex plane (private communication with Alan Palazzolo of Texas A&M). The problem discretizes a two-dimensional region of the complex plane and evaluates a specified polynomial at every discretized point. Based on the quadrant of the complex plane that the function lies in, one of four colors is assigned to that point and is plotted. Where four colors intersect, the polynomial has a root. This is a rather crude approach, but the problem is interesting in that there is immediate feedback on how effective the network implementation is because of the real-time graphics display of the complex plane mapping. The communication bottleneck occurs because of the volume of data that has to be sent through the network to the graphics board for display.

Since no complex math libraries are available in occam, simple complex add, multiply, and polynomial evaluation routines were developed. The polynomial evaluation routine uses Horners's rule for increased accuracy and performance (ref. 9). The complex math routines are listed in appendix D.

In order to implement the root visualization program on a network of transputers, first a suitable architecture must be chosen. Unlike many multiprocessors whose architecture is fixed, transputers can be wired in any configuration supported with four connections per processor.

Since the programmer must develop all of the communication routing algorithms for the network, it is currently more convenient to implement a simple, regular architecture and use simple communication procedures instead of a more complicated scheme. Additionally, the DMA link engines allow data to be piped through a network of processors with little performance penalty.

The architecture chosen for the root visualization is a pipeline. The communication buffers required for data transfer on a pipeline are only input and output buffers. An example of these along with the pipeline is shown in figure 2. If the data protocol for the network is chosen with some forethought arbitrary length pipes can be built and tested with only a software configuration parameter change.

The complex root mapper works as follows:

(1) Prompt user for region of complex plane to map
(2) Discretize specified region
(3) Compute f(z)
(4) Find quadrant of f(z)
(5) Send data to graphics board for plotting a color at point z

The graphics routines used in this test are documented in references 12 and 13. The source code for the root visualization program is listed in appendix E.

Unlike the definite integral problem, there is a large volume of data flowing through the network. For each data point to be plotted, the x-coordinate, y-coordinate, and color must be sent to the display board. The pipeline is used to compute the data and the computed data is sent back to the host processor. The host processor then sends the computed data to the graphics board. A block diagram of the network is shown in figure 12. As with the integration problem, the network architecture was chosen for its ease of implementation. A more reasonable choice for a communication bound problem would be to take advantage of as many links as possible on each transputer. One possible suggestion is shown in figure 13.

Since the size of the data required for the plotting data on the screen is constant, the order of the polynomial is varied to change the communication computation ratio on the network. All 40 processors were used for each simulation. The order of the polynomial was varied from 1 to 80 to see how much computation was required to overcome the communication bottleneck.

As with the first example, a sequential version of the program was implemented. Unlike the first problem, two processors were required for the sequential version: the computation node and the graphics board.

The next step is to determine how to divide the problem for concurrent solution on the network. Since the polynomial evaluation is direct, a known number of computations will be performed for each point in the complex plane. Dividing the two-dimensional region into strips for each processor is a simple method of distributing work and was the method used.

There are 40 processors and the screen resolution is 512 by 512 pixels. Not every processor can have an equal number of pixels to compute. The number of pixel lines (columns of pixels) each processor gets is determined as follows:

$$\text{Number of columns} = \frac{\text{Screen width}}{\text{Number of processors}}$$

and the number of processors that will have (number of columns + 1) columns is

$$\text{screen width mod number of processors}$$

with the remainder getting "number of columns" columns.

The host processor decides how to distribute the work load. The dx, and dy for the complex plane is computed along with the corresponding coordinates for the plane region of interest. Each processor gets sent a copy of the following:

```
out ! [x.min, y.min]; [dx, dy]; [x.start, x.stop];power+1::coeffs
```

where

x.min — the minimum real value of the two-dimensional region

y.min — the minimum complex value of the two-dimensional region

dx — the spacing between the x pixels

dy — the spacing between the y pixels

x.start — the local starting x coordinate for this processor

x.stop — the local stop value for the x coordinate

power — the order of the polynomial to evaluate

coeffs — the coefficients of the polynomial to evaluate

INPUT BUFFER

The input buffer for each node in the network is slightly different than the integration example presented earlier. Instead of propagating a single data set through the network, all of the data for the root solver originates from the host processor. The input buffer must be able to read the data from the host and decide whether the data is to be used locally or passed on to the next processor.

The input buffer on each processor decides where to send the data it receives by sending the first packet of data it gets to its own local node and any other data it receives to the next node in the network. This is done by setting a flag the first time data is received. The code that performs this function is shown below:

```
local := FALSE
WHILE TRUE
  SEQ
    in ? coords;du;columns;size::coeffs
  IF
    NOT local
      SEQ
        out ! coords; du; columns; size::coeffs
        local := TRUE

        through ! coords; du; columns; size::coeffs
```

where

coords[0] — the minimum real value of the two-dimensional region

coords[1] — the minimum complex value of the two-dimensional region

du[0] — the spacing between the x pixels

du[1] — the spacing between the y pixels

columns[0] — the local starting x coordinate for this processor

columns[1] — the local stop value for the x coordinate

size the order of the polynomial to evaluate

coeffs the coefficients of the polynomial to evaluate

Note that the input buffer routine is in an infinite loop. This will actually deadlock the network after all the required data has been sent; however, since it is on a network node, it does not cause any problems and the coding is easier than trying to terminate after some known number of data sets has been received. The host processor is the one which must terminate properly because if it deadlocks, the host transputer will have to be rebooted.

COMPUTE PROCESS

The compute process on the network nodes must read in the initialization data from the input buffer process and decode the information to determine what pixels to work on. Because of the way the link transfers occur, it is better to have a single long data transfer through a link rather than several short ones. The data transfer size chosen to send to the output buffer and ultimately to the host was a whole column (512) of pixels. The data format chosen was to send an array of 3(512) that contained the following information:

```
i     -- screen  x  coordinate in Integer Device Coordinates (IDC)
j     -- screen  y  coordinate in IDC
color -- color of the pixel at (i, j)
```

The size of each data transfer becomes (512)(3)(32 bits) which is 49 152 bits or 6144 (6K) bytes.

Two additional data transfer protocols were implemented to try to minimize the data transfer times through the network. The first was to change the 32-bit integers used above into 16-bit integers. This is possible since all of the data representations required can be stored in 16-bits. The second scheme was to encode the graphics information in a run-length (RL) format. Two 32-bit words were used to store the following information:

```
16-bits:  i  coordinate
16-bits:  j  coordinate
16-bits:  number of pixels to color
 8-bits:  color to use for pixels
 8-bits:  byte for direction control
```

The control byte uses the two least significant bits (LSB) to encode whether to increment or decrement the pixel drawing in the x or y-direction. The coding is as follows:

direction	LSB
+x	00
+y	01
-x	10
-y	11

One change for the RL encoded blocks that has not been implemented is to pack multiple run-length encoded blocks in to a single large block for data transfers. Right now each RL packet is sent separately so the number of channel writes for this case is significantly larger than the other two cases.

The results for the root-solver are given in the performance section below.

OUTPUT PROCESS

The output buffer process is similar to the output buffer process of the integration problem. Either of two channels is scanned using the ALT statement in occam. Any input received is sent to the adjacent node.

One addition for some of the tests was a number of internal buffers between the compute and output process and the link and the output process were used to see if they affected performance.

PERFORMANCE

All performance tests were run using a pipeline of either 40 T414 or 40 T800 transputers. Tests were performed for 1, 3, and 5 output buffers on each processor in the arrangement shown in figure 14. Polynomials of order 1, 4, 5, 10, 20, 40, and 80 were tested for each configuration.

Figure 15 shows the results for the 32-bit integer (6 kbyte block size) transfer protocol on a T414 network. The results show only a moderate performance gain for the 5-buffer case and only for the polynomial of order 10. For all other orders, the performance is approximately the same regardless of the number of buffers used.

The 16-bit (3 kbyte block size) transfer protocol results for a T414 network are shown in figure 16. The results for this case are the same for every buffer configuration. There is no advantage for the 5-buffer case with a polynomial of order 10 as there was with the 32-bit transfer protocol. The times for both the 32-bit and 16-bit transfers were nearly identical.

The results show a linear increase in solution time for polynomials of order greater than 20. Since the amount of data transferred is constant, these problems are limited by the computational speed of the processors rather than the speed limitations of the link data transfers.

The results for the run-length encoded data for a T414 network are shown in figure 17. The performance using this protocol is approximately 30 percent slower than either the 16-bit or 32-bit transfer protocol. The performance degradation is due to the number of data transfers required. The total volume of data is less than the 3 Mbytes required for the 32-bit protocol; however, the number of channel writes increases because each RL block encodes only one color.

The transputer requires only 1 μsec (20 cycles) to set up a channel communication for any size data transfer. RL encoding a whole scan-line instead of

a single color might increase the performance compared with the straight RL encoding.

The comparison involves the communication time to send a block of data versus the time to compress the block, send a smaller amount of data and decompress it. This test has not yet been performed.

The network of T800's was tested and the results are shown in figure 18 for the 32-bit integer data transfer protocol. The plot for this case is completely horizontal for polynomials from order 1 to 80. The solution time also does not change for 1, 3, or 5 return buffers. The extra floating-point performance offered by the T800 requires a considerable number of computations for every communication, ($Tc/Tp < 0.001$), in order to achieve reasonable network speedups. The current solution on the T800 network is still communication bound. Additional work causes no degradation in network performance.

SUMMARY

Parametric studies were performed to determine how to best implement direct, spatially isolated problems on transputer networks. Both computationally intensive and communication intensive problems were studied. The results indicate that the computation time per processor should exceed the communication time per processor by at least 1000 times for reasonable network performance.

APPENDIX A - SINGLE PROCESSOR SEQUENTIAL VERSION OF THE PI PROGRAM

```
+-------------------------------------------------------------------------+
|   FILE: PI_SEQ.LIS                                    SIZE:    1345 bytes |
|  SAVED: Tue Jul 12 13:38:36 1988                      PAGE:       1       |
+-------------------------------------------------------------------------+

**List of Fold**       single transputer implementation
**List of File**       PI.tsr
**List all lines
**Excluding : NO LIST folds
{{{
PROC pi.program(CHAN OF ANY keyboard, screen)
  #USE "\tdsiolib\userio.tsr"

  {{{  variables
  INT interval   :
  INT count      :
  INT dummy      :
  REAL64 delta.x :
  REAL64 sum     :
  REAL64 xi      :
  }}}
  {{{  timer variables
  TIMER time     :
  INT start.time :
  INT stop.time  :
  }}}
  {{{  misc vars and consts
  VAL r1.0   IS 1.0 (REAL64) :
  VAL r0.0   IS 0.0 (REAL64) :
  INT any :
  }}}
  SEQ
    write.full.string(screen, "Enter the number of intervals : ")
    read.echo.int(keyboard, screen, count, dummy)
    newline(screen)
    count := 1000
    sum := r0.0
    delta.x := r1.0 / (REAL64 TRUNC count)
    time ? start.time
    SEQ i = 0 FOR count
      SEQ
        xi := ((REAL64 TRUNC i) - 0.5 (REAL64)) * delta.x
        sum := sum + (delta.x * (4.0 (REAL64) / (r1.0 + (xi * xi)) ))
    time ? stop.time
    write.full.string(screen, "Time : ")
    write.int(screen, stop.time MINUS start.time, 0)
    write.full.string(screen, " low-priority ticks ")
    newline(screen)
    write.full.string(screen, "Approx. value for PI : ")
    write.real32(screen, sum, 2, 6)
    newline(screen)
    read.char(keyboard, any)
:
}}}
```

APPENDIX B - SINGLE PROCESSOR PARALLEL VERSION OF THE PI PROGRAM

```
+-------------------------------------------------------------------------+
|   FILE: PI_PAR.LIS                                  SIZE:    4980 bytes |
|  SAVED: Tue Jul 12 13:39:06 1988                    PAGE:       1       |
+-------------------------------------------------------------------------+

**List of Fold**       single transputer implementation, PARallel
**List of File**       PI2.tsr
**List all lines
**Excluding : NO LIST folds
PROC pi.program(CHAN OF ANY keyboard, screen)
  #USE "\tdsiolib\userio.tsr"

  {{{  constants
  VAL r1.0   IS 1.0 (REAL32) :
  VAL r0.0   IS 0.0 (REAL32) :
  }}}
  {{{  CHAN definitions
  VAL num.processors IS 40 :
  [num.processors]CHAN OF ANY to.network, from.network :
  [num.processors]CHAN OF ANY to.compute, from.compute :
  }}}
  {{{  PIPE buffer PROCs
  {{{  PROC input.buffer
  PROC input.buffer(CHAN OF ANY in, link.out, local.out)
    INT total.intervals :
    INT local.intervals :
    SEQ
      in ? total.intervals; local.intervals
      PAR
        link.out  ! total.intervals; local.intervals
        local.out ! total.intervals; local.intervals
  :
  }}}
  {{{  PROC return.buffer
  PROC return.buffer(CHAN OF ANY local.in, link.in, out,
                     VAL INT process.number)
    REAL32 partial.sum :
    INT loops          :
    INT proc.num :
    SEQ
      loops := 0
      WHILE loops < ((num.processors - process.number))
        ALT
          local.in ? partial.sum
            SEQ
              loops := loops + 1
              out ! process.number; partial.sum
          link.in ? proc.num; partial.sum
            SEQ
              loops := loops + 1
              out ! proc.num; partial.sum
  :
  }}}
  }}}
  {{{  END-OF-PIPE buffer PROCs
  {{{  PROC end.input.buffer, End-of-pipe
  PROC end.input.buffer(CHAN OF ANY in, local.out)
    INT total.intervals :
```

```
+-------------------------------------------------------------------------+
|   FILE: PI_PAR.LIS                                    SIZE:    4980 bytes |
|  SAVED: Tue Jul 12 13:39:06 1988                      PAGE:       2       |
+-------------------------------------------------------------------------+

    INT local.intervals :
    SEQ
      in ? total.intervals; local.intervals
      local.out ! total.intervals; local.intervals
  :
  }}}
  {{{  PROC end.return.buffer, End-of-pipe
  PROC end.return.buffer(CHAN OF ANY local.in, out, VAL INT process.number)
    REAL32 partial.sum :
    SEQ
      local.in ? partial.sum
      out ! process.number; partial.sum
  :
  }}}
  }}}
  {{{  PROC compute
  PROC compute(CHAN OF ANY in, out, VAL INT process.number)
    {{{  variables
    INT total.intervals   :
    INT local.intervals   :
    REAL32 delta.x :
    REAL32 sum     :
    REAL32 xi      :
    }}}
    {{{  computation
    SEQ
      in ? total.intervals; local.intervals
      sum := r0.0
      delta.x := r1.0 / (REAL32 TRUNC total.intervals)
      SEQ i = (process.number * local.intervals) FOR local.intervals
        SEQ
          xi := ((REAL32 TRUNC i) - 0.5 (REAL32)) * delta.x
          sum := sum + (delta.x * (4.0 (REAL32) / (r1.0 + (xi * xi)) ))
      out ! sum
    }}}
  :
  }}}
  {{{  PROC sink
  PROC sink(CHAN OF ANY in, REAL32 total)
    REAL32 result :
    INT replys:
    INT processor :
    SEQ
      replys := 0
      total := r0.0
      WHILE replys < num.processors
        SEQ
          in ? processor; result
          total := total + result
          replys := replys + 1
          {{{  COMMENT write statements
          :::A COMMENT FOLD
          {{{  write statements
```

```
+-------------------------------------------------------------------------+
|   FILE: PI_PAR.LIS                                    SIZE:    4980 bytes |
|  SAVED: Tue Jul 12 13:39:06 1988                      PAGE:       3       |
+-------------------------------------------------------------------------+

          write.full.string(screen, "total =")
          write.real32(screen, total, 2, 6)
          write.full.string(screen, "    reply from processor: ")
          write.int(screen, processor, 0)
          newline(screen)
          }}}
          }}}
  :
  }}}
  {{{  timer variables
  TIMER time     :
  INT start.time :
  INT stop.time  :
  }}}
  {{{  main program
  {{{  variables
  INT any :
  INT dummy :
  INT count :
  REAL32 sum :
  }}}
  SEQ
    {{{  print statements
    write.full.string(screen, "Enter the number of intervals per processor: ")

    read.echo.int(keyboard, screen, count, dummy)
    newline(screen)
    }}}
    {{{  main program
    time ? start.time
    PAR
      to.network[0] ! count * num.processors; count
      {{{  pipe nodes
      PAR i = 0 FOR (num.processors - 1)
        PAR
          input.buffer(to.network[i], to.network[i+1], to.compute[i])
          compute(to.compute[i], from.compute[i], i)
          return.buffer(from.compute[i], from.network[i+1], from.network[i], i
)
      }}}
      {{{  end-of-pipe node
      end.input.buffer(to.network[num.processors-1],
                       to.compute[num.processors-1])
      compute(to.compute[num.processors-1],
              from.compute[num.processors-1], num.processors - 1)
      end.return.buffer(from.compute[num.processors-1],
                        from.network[num.processors-1], num.processors - 1)
      }}}
      sink(from.network[0], sum)
    time ? stop.time
    }}}
    {{{  print statements
    write.full.string(screen, "Time : ")
```

APPENDIX B - Concluded.

```
+-------------------------------------------------------------------------------+
|   FILE: PI_PAR.LIS                                          SIZE:    4980 bytes |
|  SAVED: Tue Jul 12 13:39:06 1988                            PAGE:       4       |
+-------------------------------------------------------------------------------+

    write.int(screen, stop.time MINUS start.time, 0)
    write.full.string(screen, " low-priority ticks ")
    newline(screen)
    write.full.string(screen, "Approx. value for PI : ")
    write.real32(screen, sum, 2, 6)
    newline(screen)
    read.char(keyboard, any)
    }}}
  }}}
:
```

APPENDIX C - NETWORK VERSION OF THE PI PROGRAM

```
+-------------------------------------------------------------------------+
|   FILE: PI_EXE.LIS                                    SIZE:    2890 bytes |
|  SAVED: Thu Jun 30 13:10:50 1988                      PAGE:       1       |
+-------------------------------------------------------------------------+

**List of Fold**       network example
**List of File**       PI3.tsr
**List all lines
**Excluding : NO LIST folds
PROC pi.example(CHAN OF ANY keyboard, screen)
  #USE "\tdsiolib\userio.tsr"
  #USE "procnum.tsr"

  {{{  CHAN definitions
  {{{  channel addresses
  VAL link0.in  IS 4 :
  VAL link1.in  IS 5 :
  VAL link2.in  IS 6 :
  VAL link3.in  IS 7 :

  VAL link0.out IS 0 :
  VAL link1.out IS 1 :
  VAL link2.out IS 2 :
  VAL link3.out IS 3 :
  }}}
  CHAN OF ANY to.network, from.network :
  PLACE to.network   AT link2.out :
  PLACE from.network AT link2.in :
  }}}
  {{{  PROC sink
  PROC sink(CHAN OF ANY in, REAL32 total)
    INT replys:
    REAL32 partial.sum :
    SEQ
      replys := 0
      total := r0.0
      WHILE replys < num.processors
        SEQ
          in ? partial.sum
          total := total + partial.sum
          replys := replys + 1
          {{{  COMMENT write statements
          :::A COMMENT FOLD
          {{{  write statements
          write.full.string(screen, "partial sum: ")
          write.real32(screen, total, 2, 6)
          newline(screen)
          }}}
          }}}
  :
  }}}
  {{{  timer variables
  TIMER time     :
  INT start.time :
  INT stop.time  :
  }}}
  {{{  main program
  {{{  variables
```

```
+-------------------------------------------------------------------------+
|   FILE: PI_EXE.LIS                                    SIZE:    2890 bytes |
|  SAVED: Thu Jun 30 13:10:50 1988                      PAGE:       2       |
+-------------------------------------------------------------------------+

  INT any :
  INT dummy :
  INT count :
  REAL32 total :
  }}}
  SEQ
    {{{  print statements
    write.full.string(screen, "Enter number of intervals for each processor: "
)
    read.echo.int(keyboard, screen, count, dummy)
    newline(screen)
    }}}
    {{{  main program
    time ? start.time
    to.network ! count * num.processors; count     -- send init data to networ
k
    sink(from.network, total)
    time ? stop.time
    }}}
    {{{  print statements
    write.full.string(screen, "              Network PRI PAR communication cas
e  ")
    newline(screen)
    write.full.string(screen, "                        20MHz. T800C  ")
    newline(screen)
    newline(screen)
    write.full.string(screen, "Total number of network processors:  ")
    write.int(screen, num.processors , 0)
    newline(screen)
    write.full.string(screen, "Total number of intervals :  ")
    write.int(screen, num.processors * count, 0)
    newline(screen)
    newline(screen)
    write.full.string(screen, "Time : ")
    write.int(screen, stop.time MINUS start.time, 0)
    write.full.string(screen, " low-priority ticks ")
    write.full.string(screen, "          Time : ")
    write.real32(screen,(REAL32 ROUND (stop.time MINUS start.time)) /
                15625.0 (REAL32), 2, 4)
    write.full.string(screen, " seconds")
    newline(screen)
    newline(screen)
    write.full.string(screen, "Approximate value for PI : ")
    write.real32(screen, total, 2, 6)
    newline(screen)
    read.char(keyboard, any)
    }}}
  }}}
:
```

```
+-------------------------------------------------------------------------+
|   FILE: PI_PROG.LIS                                  SIZE:    6293 bytes |
|  SAVED: Thu Jun 30 13:11:08 1988                     PAGE:       1       |
+-------------------------------------------------------------------------+

**List of Fold**       network example
**List of File**       PI3P.tsr
**List all lines
**Excluding : NO LIST folds
{{{  SC pipe
:::A  4 10
{{{F pipe
:::F pp.tsr
PROC pipe(CHAN OF ANY from.left, to.left,
                         to.right, from.right,
                         VAL INT process.number)
  #USE "procnum.tsr"

  CHAN OF ANY input.to.compute, compute.to.output :

  {{{  PROC input.buffer
  PROC input.buffer(CHAN OF ANY link.in, link.out, local.out)
    INT total.intervals, local.intervals :
    SEQ
      link.in ? total.intervals; local.intervals
      PAR
        link.out  ! total.intervals; local.intervals
        local.out ! total.intervals; local.intervals
  :
  }}}
  {{{  PROC return.buffer
  PROC return.buffer(CHAN OF ANY local.in, link.in, link.out)
    REAL32 sum, local.sum :
    SEQ
      WHILE TRUE
        PRI ALT
          local.in ? local.sum
            link.out ! local.sum

          link.in ? sum
            link.out ! sum
  :
  }}}
  {{{  PROC compute
  PROC compute(CHAN OF ANY in, out)
    {{{  variables
    INT total.intervals   :
    INT local.intervals   :
    REAL32 delta.x :
    REAL32 sum     :
    REAL32 xi      :
    }}}
    SEQ
      in ? total.intervals; local.intervals
      sum := r0.0
      delta.x := r1.0 / (REAL32 TRUNC total.intervals)
      SEQ i = (process.number * local.intervals) FOR local.intervals
        SEQ
```

```
+-------------------------------------------------------------------------+
|   FILE: PI_PROG.LIS                                  SIZE:    6293 bytes |
|  SAVED: Thu Jun 30 13:11:08 1988                     PAGE:       2       |
+-------------------------------------------------------------------------+

          xi := ((REAL32 TRUNC i) - 0.5 (REAL32)) * delta.x
          sum := sum + (delta.x * (4.0 (REAL32) / (r1.0 + (xi * xi)) ))
      out ! sum
  :
  }}}

  [100]INT waste.space.for.proper.workspace :
  PRI PAR
    compute(input.to.compute, compute.to.output)
    PAR
      input.buffer(from.left, to.right, input.to.compute)
      return.buffer(compute.to.output, from.right, to.left)
:
}}}F
...F code
:::A  1 2
:::F pp.dcd
...F descriptor
:::A  1 4
:::F pp.dds
...F link
:::A  1 9
:::F pp.dlk
}}}

VAL B003pairs IS 1 :

{{{  CHAN definitions
{{{  channel addresses
VAL link0.in  IS 4 :
VAL link1.in  IS 5 :
VAL link2.in  IS 6 :
VAL link3.in  IS 7 :

VAL link0.out IS 0 :
VAL link1.out IS 1 :
VAL link2.out IS 2 :
VAL link3.out IS 3 :
}}}

CHAN OF ANY dummy1, dummy2 :
[8 * B003pairs]CHAN OF ANY to, from :
}}}


-- pipeline of processors, architecture f(b003pairs)
PLACED PAR
  {{{  B003pairs for pipe
  PLACED PAR j = 0 FOR (B003pairs - 1)
    PLACED PAR
      VAL i IS (8 * j) :
      PROCESSOR i T8
        PLACE to[i]      AT  link1.in  :
```

```
+-----------------------------------------------------------------------+
|  FILE: PI_PROG.LIS                                  SIZE:   6293 bytes |
| SAVED: Thu Jun 30 13:11:08 1988                     PAGE:      3       |
+-----------------------------------------------------------------------+

      PLACE from[i]    AT  link1.out :
      PLACE to[i+1]    AT  link2.out :
      PLACE from[i+1]  AT  link2.in  :
      pipe(to[i], from[i], to[i+1], from[i+1], i)
    VAL k IS (8 * j) + 1 :
    PROCESSOR k T8
      PLACE to[k]      AT  link3.in  :
      PLACE from[k]    AT  link3.out :
      PLACE to[k+1]    AT  link1.out :
      PLACE from[k+1]  AT  link1.in  :
      pipe(to[k], from[k], to[k+1], from[k+1], k)
    VAL l IS (8 * j) + 2 :
    PROCESSOR l T8
      PLACE to[l]      AT  link0.in  :
      PLACE from[l]    AT  link0.out :
      PLACE to[l+1]    AT  link2.out :
      PLACE from[l+1]  AT  link2.in  :
      pipe(to[l], from[l], to[l+1], from[l+1], l)
    PLACED PAR m = 0 FOR 2
      VAL n IS (((8 * j) + 3) + m) :
      PROCESSOR n T8
        PLACE to[n]      AT  link3.in  :
        PLACE from[n]    AT  link3.out :
        PLACE to[n+1]    AT  link2.out :
        PLACE from[n+1]  AT  link2.in  :
        pipe(to[n], from[n], to[n+1], from[n+1], n)
    VAL o IS (8 * j) + 5 :
    PROCESSOR o T8
      PLACE to[o]      AT  link3.in  :
      PLACE from[o]    AT  link3.out :
      PLACE to[o+1]    AT  link1.out :
      PLACE from[o+1]  AT  link1.in  :
      pipe(to[o], from[o], to[o+1], from[o+1], o)
    VAL p IS (8 * j) + 6 :
    PROCESSOR p T8
      PLACE to[p]      AT  link0.in  :
      PLACE from[p]    AT  link0.out :
      PLACE to[p+1]    AT  link2.out :
      PLACE from[p+1]  AT  link2.in  :
      pipe(to[p], from[p], to[p+1], from[p+1], p)
    VAL q IS (8 * j) + 7 :
    PROCESSOR q T8
      PLACE to[q]      AT  link3.in  :
      PLACE from[q]    AT  link3.out :
      PLACE to[q+1]    AT  link0.out :
      PLACE from[q+1]  AT  link0.in  :
      pipe(to[q], from[q], to[q+1], from[q+1], q)
}}}
{{{  B003pair end-of-pipe
VAL j IS (B003pairs - 1) :
PLACED PAR
  VAL i IS (8 * j) :
  PROCESSOR i T8
```

```
+-------------------------------------------------------------------------------+
|  FILE: PI_PROG.LIS                                          SIZE:   6293 bytes |
| SAVED: Thu Jun 30 13:11:08 1988                             PAGE:      4       |
+-------------------------------------------------------------------------------+

      PLACE to[i]       AT  link1.in  :
      PLACE from[i]     AT  link1.out :
      PLACE to[i+1]     AT  link2.out :
      PLACE from[i+1]   AT  link2.in  :
      pipe(to[i], from[i], to[i+1], from[i+1], i)
    VAL k IS (8 * j) + 1 :
    PROCESSOR k T8
      PLACE to[k]       AT  link3.in  :
      PLACE from[k]     AT  link3.out :
      PLACE to[k+1]     AT  link1.out :
      PLACE from[k+1]   AT  link1.in  :
      pipe(to[k], from[k], to[k+1], from[k+1], k)
    VAL l IS (8 * j) + 2 :
    PROCESSOR l T8
      PLACE to[l]       AT  link0.in  :
      PLACE from[l]     AT  link0.out :
      PLACE to[l+1]     AT  link2.out :
      PLACE from[l+1]   AT  link2.in  :
      pipe(to[l], from[l], to[l+1], from[l+1], l)
    PLACED PAR m = 0 FOR 2
      VAL n IS (((8 * j) + 3) + m) :
      PROCESSOR n T8
        PLACE to[n]       AT  link3.in  :
        PLACE from[n]     AT  link3.out :
        PLACE to[n+1]     AT  link2.out :
        PLACE from[n+1]   AT  link2.in  :
        pipe(to[n], from[n], to[n+1], from[n+1], n)
    VAL o IS (8 * j) + 5 :
    PROCESSOR o T8
      PLACE to[o]       AT  link3.in  :
      PLACE from[o]     AT  link3.out :
      PLACE to[o+1]     AT  link1.out :
      PLACE from[o+1]   AT  link1.in  :
      pipe(to[o], from[o], to[o+1], from[o+1], o)
    VAL p IS (8 * j) + 6 :
    PROCESSOR p T8
      PLACE to[p]       AT  link0.in  :
      PLACE from[p]     AT  link0.out :
      PLACE to[p+1]     AT  link2.out :
      PLACE from[p+1]   AT  link2.in  :
      pipe(to[p], from[p], to[p+1], from[p+1], p)
    VAL q IS (8 * j) + 7 :
    PROCESSOR q T8
      PLACE to[q]       AT  link3.in  :
      PLACE from[q]     AT  link3.out :
      pipe(to[q], from[q], dummy1, dummy2, q)
  }}}
```

APPENDIX D - OCCAM COMPLEX MATH PROCEDURES

```
+-------------------------------------------------------------------------+
|   FILE: COMPLEX.LIS                                  SIZE:     967 bytes  |
|  SAVED: Thu Jun 30 13:15:44 1988                     PAGE:       1        |
+-------------------------------------------------------------------------+

**List of Fold**       math
**List of File**       math.tsr
**List all lines
**Excluding : NO LIST folds
{{{  PROC cmplx.mult
PROC cmplx.mult(REAL64 x1, y1, x2, y2)
  REAL64 t.x, t.y :
  SEQ
    t.x := (x1 * x2) - (y1 * y2)
    t.y := (x1 * y2) + (x2 * y1)
    x1 := t.x                   -- return results in 1st 2 paramters
    y1 := t.y
:
}}}
{{{  PROC cmplx.add
PROC cmplx.add(REAL64 x1, y1, x2, y2)
  SEQ
    x1 := x1 + x2
    y1 := y1 + y2
:
}}}
{{{  PROC cmplx.poly
PROC cmplx.poly(REAL64 x, iy, VAL INT n, []REAL64 coeffs)
-- compute value of complex polynimial using Horner's rule
  REAL64 t.x, t.y :
  INT a :
  SEQ
    t.x := x
    t.y := iy
    cmplx.mult(t.x, t.y, coeffs[n], coeffs[n])
    SEQ i = 1 FOR (n - 1)
      SEQ
        a := n - i
        cmplx.add(t.x, t.y, coeffs[a], coeffs[a])
        cmplx.mult(t.x, t.y, x, iy)
    cmplx.add(t.x, t.y, coeffs[0], coeffs[0])
    x := t.x
    iy := t.y
:
}}}
```

APPENDIX E - ROOT VISUALIZATION PROGRAM

```
+-------------------------------------------------------------------------+
|   FILE: ROOT_EXE.LIS                                  SIZE:   11803 bytes |
|  SAVED: Thu Jun 30 13:13:10 1988                      PAGE:       1       |
+-------------------------------------------------------------------------+

**List of Fold**       root.test
**List of File**       roottest.tsr
**List all lines
**Excluding : NO LIST folds
PROC root.test(CHAN OF ANY keyboard, screen)
  #USE "\tdsiolib\userio.tsr"
  #USE "procnum.tsr"

  {{{  channel constants
  VAL link0.in  IS 4:
  VAL link1.in  IS 5:
  VAL link2.in  IS 6:
  VAL link3.in  IS 7:

  VAL link0.out IS 0:
  VAL link1.out IS 1:
  VAL link2.out IS 2:
  VAL link3.out IS 3:
  }}}
  ...F TGT graphics routines, PARTIAL        -- link2   NO LIST
  :::F TGP.tsr
  {{{  pipeline channel definitions          -- link3
  CHAN OF ANY to.pipe, from.pipe :

  PLACE to.pipe   AT link3.out :
  PLACE from.pipe AT link3.in :
  }}}

  {{{  global variables

  REAL32 x.min, x.max, y.min, y.max :
  INT order :
  [100]REAL64 coeffs :
  }}}
  {{{F PROC distribute.work
  :::F PROC02.tsr
  PROC distribute.work(CHAN OF ANY out, INT power)
    {{{  variables
    INT block.size :
    INT remainder  :
    INT x.start :
    INT x.stop  :
    REAL64 dx :
    REAL64 dy :
    }}}
    INT any :
    SEQ
      block.size := screen.width / num.processors
      remainder  := screen.width \ num.processors

      dx := REAL64 ((REAL64 (x.max - x.min)) / (REAL64 screen.width.r))
      dy := REAL64 ((REAL64 (y.max - y.min)) / (REAL64 screen.height.r))
      x.start := 1
```

```
+-------------------------------------------------------------------------+
|  FILE: ROOT_EXE.LIS                                  SIZE:   11803 bytes |
| SAVED: Thu Jun 30 13:13:10 1988                      PAGE:       2       |
+-------------------------------------------------------------------------+

      x.stop := block.size
      {{{  column size = block.size + 1
      SEQ i = 0 FOR remainder
        SEQ
          {{{  COMMENT write statements
          :::A COMMENT FOLD
          {{{  write statements
          write.full.string(screen, "*n*cx.start = ")
          write.int(screen, x.start, 0)
          write.full.string(screen, "   x.stop = ")
          write.int(screen, x.stop, 0)
          }}}
          }}}
          out ! [x.min, y.min]; [dx, dy]; [x.start, x.stop]; (power+1)::coeffs

          x.start := x.stop + 1
          x.stop := x.start + (block.size)
      }}}
--      keyboard ? any
      {{{  column size = block.size
      SEQ i = 0 FOR (num.processors - remainder)
        SEQ
          {{{  COMMENT write statements
          :::A COMMENT FOLD
          {{{  write statements
          write.full.string(screen, "*n*cx.start = ")
          write.int(screen, x.start, 0)
          write.full.string(screen, "   x.stop = ")
          write.int(screen, x.stop, 0)
          }}}
          }}}
          out ! [x.min, y.min]; [dx, dy]; [x.start, x.stop]; (power+1)::coeffs

          x.start := x.stop + 1
          x.stop := x.start + (block.size - 1)
      }}}

:
}}}F
{{{F PROC return.buffer
:::F PROC03.tsr
PROC return.buffer(CHAN OF ANY in, out, reply)
  {{{  COMMENT case: c.color.line.16
  :::A COMMENT FOLD
  {{{  case: c.color.line.16
  INT replys :
  INT size :
  INT any :
  [725][3]INT16 scan.lines :
  SEQ
    replys := 0
    WHILE replys < screen.width
      SEQ
```

```
          in ?  size::scan.lines
          out ! c.color.line.16; size::scan.lines
          reply ? any
          replys := replys + 1
    }}}
    }}}
    {{{  COMMENT case: c.RL.line
    :::A COMMENT FOLD
    {{{  case: c.RL.line
    INT rows :
    INT replys :
    INT size :
    INT any :
    [20]BYTE scan.lines :
    SEQ
      rows := 0
      WHILE rows < num.processors
        SEQ
          in ?  size::scan.lines
          IF
            size = 0
              rows := rows + 1
            TRUE
              SKIP
          out ! c.RL.line; size::scan.lines
          reply ? any
          replys := replys + 1
    }}}
    }}}
    {{{  case: c.color.line
    INT replys :
    INT size :
    INT any :
    [725][3]INT scan.lines :
    SEQ
      replys := 0
      WHILE replys < screen.width
        SEQ
          in ?  size::scan.lines
          out ! c.color.line; size::scan.lines
          reply ? any
          replys := replys + 1
    }}}
  :
  }}}F

  {{{  MAIN program
  VAL mywin IS 0 :
  {{{  variables
  TIMER time :
  INT start.time, stop.time :

  INT size :
```

```
+-------------------------------------------------------------------------+
|   FILE: ROOT_EXE.LIS                                    SIZE:   11803 bytes |
|  SAVED: Thu Jun 30 13:13:10 1988                        PAGE:       4   |
+-------------------------------------------------------------------------+

  INT reply :
  INT kchar :
  }}}
  SEQ
    init.graphics()
    {{{  COMMENT init coeffs
    :::A COMMENT FOLD
    {{{  init coeffs
    order := 8
    coeffs[0] := 0.75 (REAL64)
    coeffs[1] := -0.25 (REAL64)
    coeffs[2] :=  1.25 (REAL64)
    coeffs[3] := -2.0  (REAL64)
    coeffs[4] :=  1.0  (REAL64)

    coeffs[5] :=  11.1  (REAL64)
    coeffs[6] :=  -1.0  (REAL64)
    coeffs[7] :=  1.25  (REAL64)
    coeffs[8] :=  1.0  (REAL64)
    }}}
    }}}
    {{{  coeffs for case1.dat -- order 88
    --order := 88  -- prompt user for this for run-time sizing

    coeffs[0]  :=   2688895930118165929042780049559.83 (REAL64)
    coeffs[1]  :=   631058936978041864986597086120 3.22 (REAL64)
    coeffs[2]  :=   233939534840709935205570659516290.0 (REAL64)
    coeffs[3]  :=   5107736908815907328965093167 16931.0 (REAL64)
    coeffs[4]  :=   6.3303627503690360E+0033 (REAL64)
    coeffs[5]  :=   1.2429615635259371E+0034 (REAL64)
    coeffs[6]  :=  5.66495110305152669E+0034  (REAL64)
    coeffs[7]  :=  8.77943412511369227E+0034  (REAL64)
    coeffs[8]  :=  2.25797796733599617E+0035  (REAL64)
    coeffs[9]  :=  2.73080816039041898E+0035  (REAL64)
    coeffs[10] :=  4.66533404569097168E+0035  (REAL64)
    coeffs[11] :=  4.39969314876543267E+0035  (REAL64)
    coeffs[12] :=  5.42782242850244884E+0035  (REAL64)
    coeffs[13] :=  4.05032862074184194E+0035  (REAL64)
    coeffs[14] :=  3.81374801307389009E+0035  (REAL64)
    coeffs[15] :=  2.29606938841018777E+0035  (REAL64)
    coeffs[16] :=  1.71485373980479918E+0035  (REAL64)
    coeffs[17] :=  8.48477598392125705E+0034  (REAL64)
    coeffs[18] :=  5.17790790202715782E+0034  (REAL64)
    coeffs[19] :=  2.13869894331200797E+0034  (REAL64)
    coeffs[20] :=  1.09286839104576627E+0034  (REAL64)
    coeffs[21] :=  3.81646613218027614E+0033  (REAL64)
    coeffs[22] :=  1.66745391691839301E+0033  (REAL64)
    coeffs[23] :=  49732684225861824300079963482211 44.0      (REAL64)
    coeffs[24] :=  18918829056650812729611212105 9096.0       (REAL64)
    coeffs[25] :=  48580120082090725188431415764876.9         (REAL64)
    coeffs[26] :=  16349929288811369895712458180271.8         (REAL64)
    coeffs[27] :=  36374931481344008536087567554 46.65        (REAL64)
    coeffs[28] :=  10985751124366864548419695729 57.11        (REAL64)
```

```
+-------------------------------------------------------------------------+
|  FILE: ROOT_EXE.LIS                                  SIZE:   11803 bytes |
| SAVED: Thu Jun 30 13:13:10 1988                      PAGE:       5       |
+-------------------------------------------------------------------------+

    coeffs[29] :=  212815051948892533165452457192.527          (REAL64)
    coeffs[30] :=  584102341250508193636086178296252          (REAL64)
    coeffs[31] :=  989158904625980775784410294.281857         (REAL64)
    coeffs[32] :=  249510856509239407946462460163530          (REAL64)
    coeffs[33] :=  370559057882617439887733253.516792         (REAL64)
    coeffs[34] :=  867622892561631804288293592217909          (REAL64)
    coeffs[35] :=  113301110378500700129189166270315          (REAL64)
    coeffs[36] :=  248397750292467888668978900207246          (REAL64)
    coeffs[37] :=  285859983602747700127993.727648760         (REAL64)
    coeffs[38] :=  591308295697789083173941575184057          (REAL64)
    coeffs[39] :=  600835092169430365039726713586514          (REAL64)
    coeffs[40] :=  118040506982671898406089309743975          (REAL64)
    coeffs[41] :=  106079400171860179627.222476776953         (REAL64)
    coeffs[42] :=  19906543222849697093.9861423290810         (REAL64)
    coeffs[43] :=  1584447790602882067.48812525032395         (REAL64)
    coeffs[44] :=  285401472996503223.456288831832534         (REAL64)
    coeffs[45] :=  20143945845636738.0824270081621489         (REAL64)
    coeffs[46] :=  3497473470335931.06551666661023121         (REAL64)
    coeffs[47] :=  219112572495724.945212388655304123         (REAL64)
    coeffs[48] :=  36801087774640.0061595353794202558         (REAL64)
    coeffs[49] :=  2047853441552.65983127088214246812         (REAL64)
    coeffs[50] :=  333731879974.996810959590064780248         (REAL64)
    coeffs[51] :=  16501719098.3088148178255474997398         (REAL64)
    coeffs[52] :=  2616121894.8896689274116817180519 6        (REAL64)
    coeffs[53] :=  114947347.068238925297516034243685         (REAL64)
    coeffs[54] :=  17766959.56546203790204064428727 35        (REAL64)
    coeffs[55] :=  693434.64653022571164903639132779 9        (REAL64)
    coeffs[56] :=  104692.61394756764294516860284138 6        (REAL64)
    coeffs[57] :=  3626.60816715194259217045895894494         (REAL64)
    coeffs[58] :=  535.66739423560287116461629067122 8        (REAL64)
    coeffs[59] :=  16.44669026654932685618926938749 06        (REAL64)
    coeffs[60] :=  2.37979624754459180456962481262258         (REAL64)
    coeffs[61] :=  6.463281181801569101E-0002  (REAL64)
    coeffs[62] :=  9.172209135671340010E-0003  (REAL64)
    coeffs[63] :=  2.197458385268279111E-0004  (REAL64)
    coeffs[64] :=  3.061382571345561467E-0005  (REAL64)
    coeffs[65] :=  6.446212248480929382E-0007  (REAL64)
    coeffs[66] :=  8.823250234915350103E-0008  (REAL64)
    coeffs[67] :=  1.625075783878327454E-0009  (REAL64)
    coeffs[68] :=  2.186853501421395672E-0010  (REAL64)
    coeffs[69] :=  3.501254030114446424E-0012  (REAL64)
    coeffs[70] :=  4.634883471937603426E-0013  (REAL64)
    coeffs[71] :=  6.398877449419483224E-0015  (REAL64)
    coeffs[72] :=  8.336696149052224707E-0016  (REAL64)
    coeffs[73] :=  9.820986584730996429E-0018  (REAL64)
    coeffs[74] :=  1.259768369772852991E-0018  (REAL64)
    coeffs[75] :=  1.248830190534017194E-0020  (REAL64)
    coeffs[76] :=  1.577699371679007549E-0021  (REAL64)
    coeffs[77] :=  1.291500436521210235E-0023  (REAL64)
    coeffs[78] :=  1.607363897472892237E-0024  (REAL64)
    coeffs[79] :=  1.058123161156898495E-0026  (REAL64)
    coeffs[80] :=  1.297613499647725398E-0027  (REAL64)
    coeffs[81] :=  6.605231009053681181E-0030  (REAL64)
```

```
+-------------------------------------------------------------------------+
|   FILE: ROOT_EXE.LIS                                  SIZE:   11803 bytes |
|  SAVED: Thu Jun 30 13:13:10 1988                      PAGE:       6       |
+-------------------------------------------------------------------------+

    coeffs[82] :=  7.98284225394925304E-0031  (REAL64)
    coeffs[83] :=  2.95026139166056635E-0033  (REAL64)
    coeffs[84] :=  3.51434333271103682E-0034  (REAL64)
    coeffs[85] :=  8.39451131239523707E-0037  (REAL64)
    coeffs[86] :=  9.85671254320613681E-0038  (REAL64)
    coeffs[87] :=  1.14289663220184123E-0040 (REAL64)
    coeffs[88] :=  1.32288603944358394E-0041  (REAL64)
    }}}
    {{{  prompt user for screen coordinate data
    write.full.string(screen, "Enter Real min:      ")
    read.echo.real32(keyboard, screen, x.min, kchar)
    write.full.string(screen, "Enter Real max:      ")
    read.echo.real32(keyboard, screen, x.max, kchar)
    write.full.string(screen, "Enter Imaginary min: ")
    read.echo.real32(keyboard, screen, y.min, kchar)
    write.full.string(screen, "Enter Imaginary max: ")
    read.echo.real32(keyboard, screen, y.max, kchar)
    write.full.string(screen, "Enter order of polynomial: ")
    read.echo.int(keyboard, screen, order, kchar)
    }}}
    time ? start.time  -- after user inputs data....
    {{{  init windows/viewport
    set.window.2d(x.min, y.min, x.max, y.max, mywin)
    set.viewport.2d(0.0 (REAL32), 0.0 (REAL32), r1.0, r1.0, mywin)
    {{{  set screen
    g.send2(c.select.screen, 0)
    g.send2(c.clear.screen, 0)
    g.send2(c.display.screen, 0)
    }}}
    activate.viewport.2d(mywin)
    clear.window(0)
    display.viewport.2d(mywin)
    }}}
    {{{  distribute work and read results
    PAR
      distribute.work(to.pipe, order)
      return.buffer(from.pipe, to.graphic, from.graphic)
    }}}
    finit.graphics()
    time ? stop.time
    {{{  write results
    write.full.string(screen, "Low priority ticks: ")
    write.int(screen, (stop.time MINUS start.time), 0)
    keyboard ? reply
    }}}
  }}}
:
```

```
+-------------------------------------------------------------------------+
|   FILE: ROOT_PRG.LIS                                SIZE:   19688 bytes |
|  SAVED: Thu Jun 30 13:14:28 1988                    PAGE:       1       |
+-------------------------------------------------------------------------+

**List of Fold**       pipe program
**List of File**       pipe00.tsr
**List all lines
**Excluding : NO LIST folds
{{{  SC pipe node
:::A  3 10
{{{F pipe node
:::F pipe.tsr
PROC pipe(CHAN OF ANY from.left, to.left,
                     to.right, from.right, dummy)
  {{{F PROC input
  :::F PROC00.tsr
  PROC input(CHAN OF ANY in, out, through)
    {{{  variables
    [100]REAL64 coeffs :
    [2]REAL64 du :
    [2]REAL32 coords :
    [2]INT columns :

    INT size :
    }}}
    BOOL local :
    SEQ
      local := FALSE
      WHILE TRUE
        SEQ
          in ? coords; du; columns; size::coeffs
          IF
            NOT local
              SEQ
                out ! coords; du; columns; size::coeffs
                local := TRUE
            TRUE
              through ! coords; du; columns; size::coeffs
  :
  }}}F
  {{{F PROC output -- replicated ALT
  :::F PROC01.tsr
  PROC output([2]CHAN OF ANY in, CHAN OF ANY out)
    {{{  COMMENT round-robin ALT : requires [2] CHAN OF ANY in
    :::A COMMENT FOLD
    {{{  round-robin ALT : requires [2] CHAN OF ANY in
    INT size :
    --[725][3]INT scan.lines :
    --[725][3]INT16 scan.lines :
    --[200]BYTE scan.lines :
    INT count :
    SEQ
      count := 0
      WHILE TRUE
        SEQ
          ALT i = count FOR (SIZE in)
            in[i\(SIZE in)] ? size::scan.lines
```

```
+-------------------------------------------------------------------------+
|   FILE: ROOT_PRG.LIS                                  SIZE:   19688 bytes |
|  SAVED: Thu Jun 30 13:14:28 1988                      PAGE:       2       |
+-------------------------------------------------------------------------+

              SEQ
                out ! size::scan.lines
                count := count PLUS 1
    }}}
    }}}
    {{{  "regular" ALT
    INT size :
    [725][3]INT scan.lines :
    --[725][3]INT16 scan.lines :
    --[200]BYTE scan.lines :
    INT count :
    SEQ
      count := 0
      WHILE TRUE
        SEQ
          ALT i = 0 FOR (SIZE in)
            in[i] ? size::scan.lines
              out ! size::scan.lines
    }}}
  :
  }}}F
  {{{  COMMENT PROC output --   non-replicated ALT
  :::A COMMENT FOLD
  {{{  PROC output --   non-replicated ALT
  PROC output(CHAN OF ANY in1, in2, CHAN OF ANY out)
    {{{  COMMENT round-robin ALT : requires [2] CHAN OF ANY in
    :::A COMMENT FOLD
    {{{  round-robin ALT : requires [2] CHAN OF ANY in
    INT size :
    --[725][3]INT scan.lines :
    --[725][3]INT16 scan.lines :
    --[200]BYTE scan.lines :
    INT count :
    SEQ
      count := 0
      WHILE TRUE
        SEQ
          ALT i = count FOR (SIZE in)
            in[i\(SIZE in)] ? size::scan.lines
              SEQ
                out ! size::scan.lines
                count := count PLUS 1
    }}}
    }}}
    {{{  "regular" ALT
    INT size :
    [725][3]INT scan.lines :
    --[725][3]INT16 scan.lines :
    --[200]BYTE scan.lines :
    INT count :
    SEQ
      count := 0
      WHILE TRUE
```

```
        SEQ
          ALT
            in1 ? size::scan.lines
              out ! size::scan.lines
            in2 ? size::scan.lines
              out ! size::scan.lines
    }}}
  :
  }}}
  }}}
  {{{F PROC compute
  :::F PROC.tsr
  PROC compute(CHAN OF ANY in, out)
    {{{  COMMENT case: c.color.line.16
    :::A COMMENT FOLD
    {{{  case: c.color.line.16
    {{{  PROC cmplx.mult
    PROC cmplx.mult(REAL64 x1, y1, x2, y2)
      REAL64 t.x, t.y :
      SEQ
        t.x := (x1 * x2) - (y1 * y2)
        t.y := (x1 * y2) + (x2 * y1)
        x1 := t.x                    -- return results in 1st 2 paramters
        y1 := t.y
    :
    }}}
    {{{  PROC cmplx.add
    PROC cmplx.add(REAL64 x1, y1, x2, y2)
      SEQ
        x1 := x1 + x2
        y1 := y1 + y2
    :
    }}}
    {{{  PROC cmplx.poly
    PROC cmplx.poly(REAL64 x, iy, VAL INT n, []REAL64 coeffs)
    -- compute value of complex polynimial using Horner's rule
      REAL64 t.x, t.y :
      INT a :
      SEQ
        t.x := x
        t.y := iy
        cmplx.mult(t.x, t.y, coeffs[n], coeffs[n])
        SEQ i = 1 FOR (n - 1)
          SEQ
            a := n - i
            cmplx.add(t.x, t.y, coeffs[a], coeffs[a])
            cmplx.mult(t.x, t.y, x, iy)
        cmplx.add(t.x, t.y, coeffs[0], coeffs[0])
        x := t.x
        iy := t.y
    :
    }}}
```

```
+-------------------------------------------------------------------------------+
|   FILE: ROOT_PRG.LIS                                  SIZE:   19688 bytes     |
|  SAVED: Thu Jun 30 13:14:28 1988                      PAGE:       4           |
+-------------------------------------------------------------------------------+

    {{{  variables and constants
    INT size :
    [100]REAL64 coeffs :

    [725][3]INT16 scan.lines :
    REAL64 x, y :
    REAL64 x.old, y.old :

    [2]REAL32 coords :
    [2]REAL64 du :
    [2]INT columns :

    {{{  define color registers
    VAL black  IS  0 :                    -- define some color register numb
ers
    VAL red    IS 31 :
    VAL green  IS 47 :
    VAL blue   IS 63 :
    VAL yellow IS 79 :

    INT color :
    }}}
    VAL screen.height IS 512 :
    }}}
    SEQ
      --out ! [x.min, y.min]; [dx, dy]; [x.start, x.stop]; power::coeffs
      in ? coords; du; columns; size::coeffs
      x := (REAL64 coords[0]) + (du[0] * (REAL64 TRUNC columns[0]))
      {{{  compute the rows
      SEQ i = columns[0] FOR ((columns[1] - columns[0]) + 1)
        SEQ
          y := REAL64 coords[1]  --y.min
          {{{  compute the column
          SEQ j = 0 FOR screen.height
            SEQ
              x.old := x
              y.old := y
              cmplx.poly(x, y, size - 1, coeffs)
              {{{  compute quadrant, assign color
              IF
                x >= 0.0 (REAL64)
                  IF
                    y >= 0.0 (REAL64)
                      color := yellow
                    TRUE
                      color := red
                TRUE
                  IF
                    y >= 0.0 (REAL64)
                      color := green
                    TRUE
                      color := blue
              }}}
```

```
+-------------------------------------------------------------------------+
|  FILE: ROOT_PRG.LIS                                  SIZE:   19688 bytes |
| SAVED: Thu Jun 30 13:14:28 1988                      PAGE:       5       |
+-------------------------------------------------------------------------+

            x := x.old
            y := y.old
            scan.lines[j][0] := INT16 i
            scan.lines[j][1] := INT16 j
            scan.lines[j][2] := INT16 color
            y := y + du[1]
        }}}
        x := x + du[0]
        out ! screen.height::scan.lines
    }}}
  }}}
  }}}
  {{{  COMMENT case: c.RL.line
  :::A COMMENT FOLD
  {{{  case: c.RL.line
  {{{  PROC cmplx.mult
  PROC cmplx.mult(REAL64 x1, y1, x2, y2)
    REAL64 t.x, t.y :
    SEQ
      t.x := (x1 * x2) - (y1 * y2)
      t.y := (x1 * y2) + (x2 * y1)
      x1 := t.x                  -- return results in 1st 2 paramters
      y1 := t.y
  :
  }}}
  {{{  PROC cmplx.add
  PROC cmplx.add(REAL64 x1, y1, x2, y2)
    SEQ
      x1 := x1 + x2
      y1 := y1 + y2
  :
  }}}
  {{{  PROC cmplx.poly
  PROC cmplx.poly(REAL64 x, iy, VAL INT n, []REAL64 coeffs)
  -- compute value of complex polynimial using Horner's rule
    REAL64 t.x, t.y :
    INT a :
    SEQ
      t.x := x
      t.y := iy
      cmplx.mult(t.x, t.y, coeffs[n], coeffs[n])
      SEQ i = 1 FOR (n - 1)
        SEQ
          a := n - i
          cmplx.add(t.x, t.y, coeffs[a], coeffs[a])
          cmplx.mult(t.x, t.y, x, iy)
      cmplx.add(t.x, t.y, coeffs[0], coeffs[0])
      x := t.x
      iy := t.y
  :
  }}}

  {{{  variables and constants
```

```
+-------------------------------------------------------------------------+
|   FILE: ROOT_PRG.LIS                                  SIZE:   19688 bytes |
|  SAVED: Thu Jun 30 13:14:28 1988                      PAGE:       6       |
+-------------------------------------------------------------------------+

    INT size :
    [100]REAL64 coeffs :

    REAL64 x, y :
    REAL64 x.old, y.old :

    [2]REAL32 coords :
    [2]REAL64 du :
    [2]INT columns :

    {{{  define color registers
    VAL black  IS  0 :                      -- define some color register numb
ers
    VAL red    IS 31 :
    VAL green  IS 47 :
    VAL blue   IS 63 :
    VAL yellow IS 79 :

    INT color :
    }}}
    VAL screen.height IS 512 :
    }}}
    {{{  abbreviations
    VAL bpw IS 4 :
    [4]INT line.buffer :
    [4 * bpw]BYTE pixel.buffer RETYPES line.buffer : -- done this way for alig
nment

    INT16 i.16    RETYPES [pixel.buffer FROM 0 FOR 2] :
    INT16 j.16    RETYPES [pixel.buffer FROM 2 FOR 2] :
    INT16 count   RETYPES [pixel.buffer FROM 4 FOR 2] :
    BYTE  colour  IS pixel.buffer[6] :
    BYTE  control IS pixel.buffer[7] :
    }}}
    SEQ
      control := #01 (BYTE)  -- + y direction
      --out ! [x.min, y.min]; [dx, dy]; [x.start, x.stop]; power::coeffs
      in ? coords; du; columns; size::coeffs
      {{{  compute the rows
      x := (REAL64 coords[0]) + (du[0] * (REAL64 TRUNC columns[0]))
      SEQ i = columns[0] FOR ((columns[1] - columns[0]) + 1)
        INT j :
        INT color.start :
        SEQ
          y := REAL64 coords[1]  --y.min
          j := 0
          {{{  compute the column
          WHILE j < screen.height
            SEQ
              {{{  get color for starting point
              x.old := x
              y.old := y
              cmplx.poly(x, y, size - 1, coeffs)  -- get first color for loopi
```

```
+-------------------------------------------------------------------------+
|   FILE: ROOT_PRG.LIS                                  SIZE:   19688 bytes |
| SAVED: Thu Jun 30 13:14:28 1988                       PAGE:       7       |
+-------------------------------------------------------------------------+
```

ng

```
                {{{  compute quadrant, assign color
                IF
                  x >= 0.0 (REAL64)
                    IF
                      y >= 0.0 (REAL64)
                        color := yellow
                      TRUE
                        color := red
                  TRUE
                    IF
                      y >= 0.0 (REAL64)
                        color := green
                      TRUE
                        color := blue
                }}}
                x := x.old
                y := y.old
                }}}
                {{{  init variables for this column
                i.16 := INT16 i  -- starting point
                j.16 := INT16 j
                count := 0 (INT16)      -- init count
                color.start := color
                colour := BYTE color
                }}}
                WHILE (color = color.start) AND (j < screen.height)
                  SEQ
                    {{{  increment variables
                    y := y + du[1]   -- increment by dy
                    j := j + 1
                    count := count + (1 (INT16))
                    }}}
                    {{{  get color for next y
                    x.old := x
                    y.old := y
                    cmplx.poly(x, y, size - 1, coeffs)
                    {{{  compute quadrant, assign color
                    IF
                      x >= 0.0 (REAL64)
                        IF
                          y >= 0.0 (REAL64)
                            color := yellow
                          TRUE
                            color := red
                      TRUE
                        IF
                          y >= 0.0 (REAL64)
                            color := green
                          TRUE
                            color := blue
                    }}}
                    x := x.old
```

```
+-------------------------------------------------------------------------+
|   FILE: ROOT_PRG.LIS                                  SIZE:   19688 bytes |
|  SAVED: Thu Jun 30 13:14:28 1988                      PAGE:       8       |
+-------------------------------------------------------------------------+

                  y := y.old
                  }}}
              out ! 8::pixel.buffer       -- this won't work for d700d
          }}}
          x := x + du[0]
      }}}
      out ! 0::pixel.buffer  --signal end of column
    }}}
    }}}
    {{{  case: c.color.line
    {{{  PROC cmplx.mult
    PROC cmplx.mult(REAL64 x1, y1, x2, y2)
      REAL64 t.x, t.y :
      SEQ
        t.x := (x1 * x2) - (y1 * y2)
        t.y := (x1 * y2) + (x2 * y1)
        x1 := t.x                  -- return results in 1st 2 paramters
        y1 := t.y
    :
    }}}
    {{{  PROC cmplx.add
    PROC cmplx.add(REAL64 x1, y1, x2, y2)
      SEQ
        x1 := x1 + x2
        y1 := y1 + y2
    :
    }}}
    {{{  PROC cmplx.poly
    PROC cmplx.poly(REAL64 x, iy, VAL INT n, []REAL64 coeffs)
    -- compute value of complex polynimial using Horner's rule
      REAL64 t.x, t.y :
      INT a :
      SEQ
        t.x := x
        t.y := iy
        cmplx.mult(t.x, t.y, coeffs[n], coeffs[n])
        SEQ i = 1 FOR (n - 1)
          SEQ
            a := n - i
            cmplx.add(t.x, t.y, coeffs[a], coeffs[a])
            cmplx.mult(t.x, t.y, x, iy)
        cmplx.add(t.x, t.y, coeffs[0], coeffs[0])
        x := t.x
        iy := t.y
    :
    }}}

    {{{  variables and constants
    INT size :
    [100]REAL64 coeffs :

    [725][3]INT scan.lines :
    REAL64 x, y :
```

```
+-------------------------------------------------------------------------------+
|   FILE: ROOT_PRG.LIS                                    SIZE:   19688 bytes   |
|  SAVED: Thu Jun 30 13:14:28 1988                        PAGE:       9         |
+-------------------------------------------------------------------------------+

    REAL64 x.old, y.old :

    [2]REAL32 coords :
    [2]REAL64 du :
    [2]INT columns :

    {{{  define color registers
    VAL black  IS  0 :                      -- define some color register numb
ers
    VAL red    IS 31 :
    VAL green  IS 47 :
    VAL blue   IS 63 :
    VAL yellow IS 79 :

    INT color :
    }}}
    VAL screen.height IS 512 :
    }}}
    SEQ
      --out ! [x.min, y.min]; [dx, dy]; [x.start, x.stop]; power::coeffs
      in ? coords; du; columns; size::coeffs
      x := (REAL64 coords[0]) + (du[0] * (REAL64 TRUNC columns[0]))
      {{{  compute the rows
      SEQ i = columns[0] FOR ((columns[1] - columns[0]) + 1)
        SEQ
          y := REAL64 coords[1]  --y.min
          {{{  compute the column
          SEQ j = 0 FOR screen.height
            SEQ
              x.old := x
              y.old := y
              cmplx.poly(x, y, size - 1, coeffs)
              {{{  compute quadrant, assign color
              IF
                x >= 0.0 (REAL64)
                  IF
                    y >= 0.0 (REAL64)
                      color := yellow
                    TRUE
                      color := red
                TRUE
                  IF
                    y >= 0.0 (REAL64)
                      color := green
                    TRUE
                      color := blue
              }}}
              x := x.old
              y := y.old
              scan.lines[j][0] := i
              scan.lines[j][1] := j
              scan.lines[j][2] := color
              y := y + du[1]
```

```
+-------------------------------------------------------------------------+
|   FILE: ROOT_PRG.LIS                                  SIZE:   19688 bytes |
|  SAVED: Thu Jun 30 13:14:28 1988                      PAGE:      10       |
+-------------------------------------------------------------------------+

          }}}
          x := x + du[0]
          out ! screen.height::scan.lines
      }}}
    }}}
  :
  }}}F
  {{{F PROC thru.buffer
  :::F PROC04.tsr
  PROC thru.buffer(CHAN OF ANY local.in, local.out)
    INT size :
    [725][3]INT scan.lines :
    --[725][3]INT16 scan.lines :
    --[200]BYTE scan.lines :
    WHILE TRUE
      SEQ
        local.in ? size::scan.lines
        local.out ! size::scan.lines
  :
  }}}F


  CHAN OF ANY to.compute, from.compute :
  [2]CHAN OF ANY to.buffer :
  [2]CHAN OF ANY thru :
  PRI PAR
    PAR
      input(from.left, to.compute, to.right)
      thru.buffer(from.compute, to.buffer[0])
      thru.buffer(to.buffer[0], thru[0])
      thru.buffer(from.right, to.buffer[1])
      thru.buffer(to.buffer[1], thru[1])
      output(thru , to.left)
      --output(from.right, from.compute , to.left)
    compute(to.compute, from.compute)
:
}}}F
...F code
:::A  1 2
:::F pipe.dcd
...F descriptor
:::A  1 4
:::F pipe.dds
...F link
:::A  1 9
:::F pipe.dlk
}}}
{{{  SC b007 board
:::A  3 10
{{{F b007 board
:::F b007.tsr
PROC graphics(CHAN OF ANY to, from, load.link)
  #USE "\d700c\graphlib\b007lib.tsr"
```

```
+-------------------------------------------------------------------------+
|   FILE: ROOT_PRG.LIS                                  SIZE:  19688 bytes |
|  SAVED: Thu Jun 30 13:14:28 1988                      PAGE:     11       |
+-------------------------------------------------------------------------+

  B007(to, from)
:
}}}F
...F code
:::A  1 2
:::F b007.dcd
...F descriptor
:::A  1 4
:::F b007.dds
...F link
:::A  1 9
:::F b007.dlk
}}}

VAL B003pairs IS 5 :

{{{  CHAN definitions
{{{  channel addresses
VAL link0.in  IS 4 :
VAL link1.in  IS 5 :
VAL link2.in  IS 6 :
VAL link3.in  IS 7 :

VAL link0.out IS 0 :
VAL link1.out IS 1 :
VAL link2.out IS 2 :
VAL link3.out IS 3 :
}}}

CHAN OF ANY dummy1, dummy2 , b007.boot :
[8 * B003pairs]CHAN OF ANY to, from , dummy :

CHAN OF ANY to.graphics, from.graphics :
}}}


-- pipeline of processors, architecture f(b003pairs)
PLACED PAR
  {{{  B003pairs for pipe
  PLACED PAR j = 0 FOR (B003pairs - 1)
    PLACED PAR
      VAL i IS (8 * j) :
      PROCESSOR i T8
        PLACE to[i]      AT  link1.in  :
        PLACE from[i]    AT  link1.out :
        PLACE to[i+1]    AT  link2.out :
        PLACE from[i+1]  AT  link2.in  :
        pipe(to[i], from[i], to[i+1], from[i+1], dummy[i])
      VAL k IS (8 * j) + 1 :
      PROCESSOR k T8
        PLACE to[k]      AT  link3.in  :
        PLACE from[k]    AT  link3.out :
        PLACE to[k+1]    AT  link1.out :
```

```
FILE: ROOT_PRG.LIS                                    SIZE:   19688 bytes
SAVED: Thu Jun 30 13:14:28 1988                       PAGE:      12
```

```
      PLACE from[k+1]  AT  link1.in  :
      pipe(to[k], from[k], to[k+1], from[k+1], dummy[k])
    VAL l IS (8 * j) + 2 :
    PROCESSOR l T8
      PLACE to[l]      AT  link0.in  :
      PLACE from[l]    AT  link0.out :
      PLACE to[l+1]    AT  link2.out :
      PLACE from[l+1]  AT  link2.in  :
      pipe(to[l], from[l], to[l+1], from[l+1], dummy[l])
    PLACED PAR m = 0 FOR 2
      VAL n IS (((8 * j) + 3) + m) :
      PROCESSOR n T8
        PLACE to[n]      AT  link3.in  :
        PLACE from[n]    AT  link3.out :
        PLACE to[n+1]    AT  link2.out :
        PLACE from[n+1]  AT  link2.in  :
        pipe(to[n], from[n], to[n+1], from[n+1], dummy[n])
    VAL o IS (8 * j) + 5 :
    PROCESSOR o T8
      PLACE to[o]      AT  link3.in  :
      PLACE from[o]    AT  link3.out :
      PLACE to[o+1]    AT  link1.out :
      PLACE from[o+1]  AT  link1.in  :
      pipe(to[o], from[o], to[o+1], from[o+1], dummy[o])
    VAL p IS (8 * j) + 6 :
    PROCESSOR p T8
      PLACE to[p]      AT  link0.in  :
      PLACE from[p]    AT  link0.out :
      PLACE to[p+1]    AT  link2.out :
      PLACE from[p+1]  AT  link2.in  :
      pipe(to[p], from[p], to[p+1], from[p+1], dummy[p])
    VAL q IS (8 * j) + 7 :
    PROCESSOR q T8
      PLACE to[q]      AT  link3.in  :
      PLACE from[q]    AT  link3.out :
      PLACE to[q+1]    AT  link0.out :
      PLACE from[q+1]  AT  link0.in  :
      pipe(to[q], from[q], to[q+1], from[q+1], dummy[q])
}}}
{{{  B003pair end-of-pipe
VAL j IS (B003pairs - 1) :
PLACED PAR
  VAL i IS (8 * j) :
  PROCESSOR i T8
    PLACE to[i]      AT  link1.in  :
    PLACE from[i]    AT  link1.out :
    PLACE to[i+1]    AT  link2.out :
    PLACE from[i+1]  AT  link2.in  :
    pipe(to[i], from[i], to[i+1], from[i+1], dummy[i])
  VAL k IS (8 * j) + 1 :
  PROCESSOR k T8
    PLACE to[k]      AT  link3.in  :
    PLACE from[k]    AT  link3.out :
```

```
+-------------------------------------------------------------------------------+
|   FILE: ROOT_PRG.LIS                                    SIZE:   19688 bytes   |
| SAVED: Thu Jun 30 13:14:28 1988                         PAGE:      13         |
+-------------------------------------------------------------------------------+

      PLACE to[k+1]    AT  link1.out :
      PLACE from[k+1]  AT  link1.in  :
      pipe(to[k], from[k], to[k+1], from[k+1], dummy[k])
    VAL l IS (8 * j) + 2 :
    PROCESSOR l T8
      PLACE to[l]      AT  link0.in  :
      PLACE from[l]    AT  link0.out :
      PLACE to[l+1]    AT  link2.out :
      PLACE from[l+1]  AT  link2.in  :
      pipe(to[l], from[l], to[l+1], from[l+1], dummy[l])
    PLACED PAR m = 0 FOR 2
      VAL n IS (((8 * j) + 3) + m) :
      PROCESSOR n T8
        PLACE to[n]      AT  link3.in  :
        PLACE from[n]    AT  link3.out :
        PLACE to[n+1]    AT  link2.out :
        PLACE from[n+1]  AT  link2.in  :
        pipe(to[n], from[n], to[n+1], from[n+1], dummy[n])
    VAL o IS (8 * j) + 5 :
    PROCESSOR o T8
      PLACE to[o]      AT  link3.in  :
      PLACE from[o]    AT  link3.out :
      PLACE to[o+1]    AT  link1.out :
      PLACE from[o+1]  AT  link1.in  :
      pipe(to[o], from[o], to[o+1], from[o+1], dummy[o])
    VAL p IS (8 * j) + 6 :
    PROCESSOR p T8
      PLACE to[p]      AT  link0.in  :
      PLACE from[p]    AT  link0.out :
      PLACE to[p+1]    AT  link2.out :
      PLACE from[p+1]  AT  link2.in  :
      pipe(to[p], from[p], to[p+1], from[p+1], dummy[p])
    VAL q IS (8 * j) + 7 :
    PROCESSOR q T8
      PLACE to[q]      AT  link3.in  :
      PLACE from[q]    AT  link3.out :
      PLACE b007.boot  AT  link0.out :
      pipe(to[q], from[q], dummy1, dummy2, b007.boot)
  }}}
  {{{  graphics board
  PROCESSOR 999 T8
    PLACE to.graphics AT link1.in :
    PLACE from.graphics AT link1.out :
    PLACE b007.boot     AT link0.in :
    graphics(to.graphics, from.graphics, b007.boot)
  }}}
```

REFERENCES

1. Babb, R.G. II, ed.: Programming Parallel Processors. Addison-Wesley, 1987.

2. Homewood, M., et al.: The IMS T800 Transputer. IEEE Micro, vol. 7, no. 5, Oct. 1987, pp. 10-26.

3. May, D.; and Taylor, R.: OCCAM - An Overview. Microprocessors and Microsystems, vol. 8, no. 2, Mar. 1984, pp. 73-79.

4. Transputer Development System User Manual. INMOS Corp., Colorado Springs, CO.

5. May, D.; and Shepherd, R.: The Transputer Implementation of OCCAM. Fifth Generation Computer Systems 1984, Elsevier North Holland, 1984, pp.533-541.

6. May, D.: OCCAM. SIGPLAN Notices, vol. 18, no. 4, Apr. 1983, pp. 69-79.

7. T414 Engineering Data. INMOS Corp., Colorado Springs, CO.

8. Atkin, P.: Performance Maximization. Technical Note 17, INMOS Corp., Colorado Springs, CO.

9. Carnahan, B.; Luther, H.A.; and Wilkes, J.O.: Applied Numerical Methods. John Wiley and Sons, 1969.

10. Danial, A.; and Watson, J.: Iterative Finite Element Solver on Transputer Networks. Lewis Structures Technology 1988, Vol. 1 - Structural Dynamics, NASA CP-3003-VOL-1, 1988, pp. 113-123.

11. Gusfaston, J.L.; Montry, G.R.; and Benner, R.E.: Development of Parallel Methods for a 1024-Processor Hypercube. SIAM J. Scientific Stat. Comput., vol. 9, no. 4., July 1988, pp. 609-638.

12. Ellis, G.K.: Two-Dimensional Graphics Tools for a Transputer Based Display Board. NASA TM-100820, 1988.

13. Ellis, G.K.: User Manual for the Two-Dimensional Transputer Graphics Toolkit. NASA TM-100974, 1988.

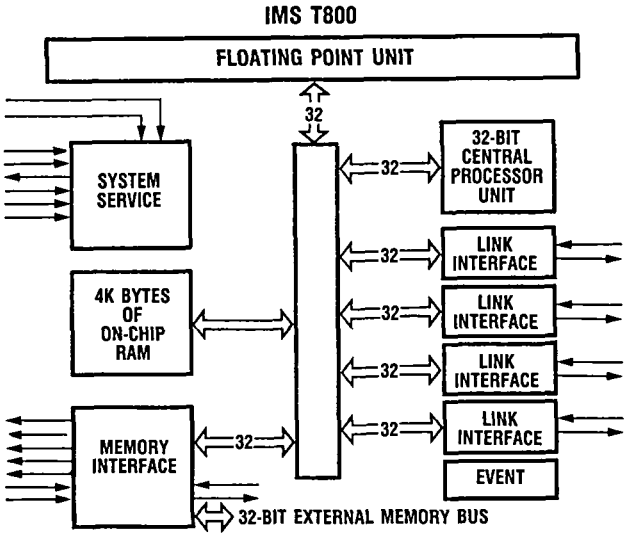

FIGURE 1. - BLOCK DIAGRAM OF A T800 FLOATING POINT TRANSPUTER.

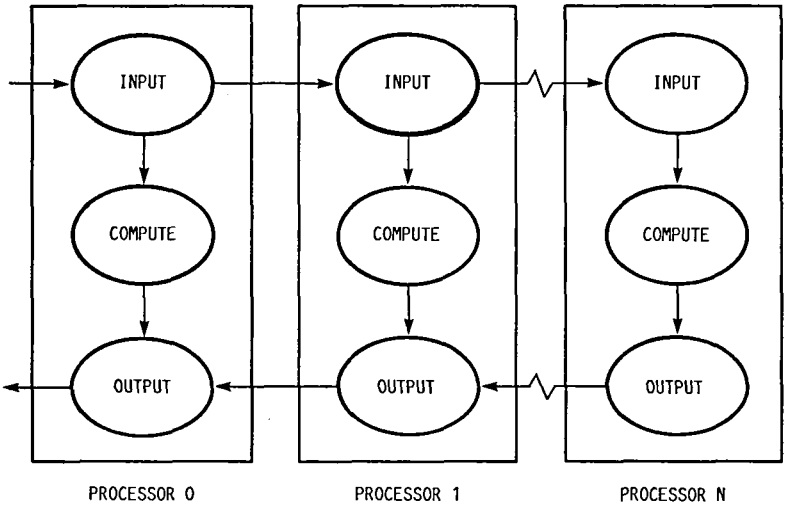

FIGURE 2. - PIPELINE OF TRANSPUTERS SHOWING INPUT, OUTPUT, AND COMPUTE BUFFERS.

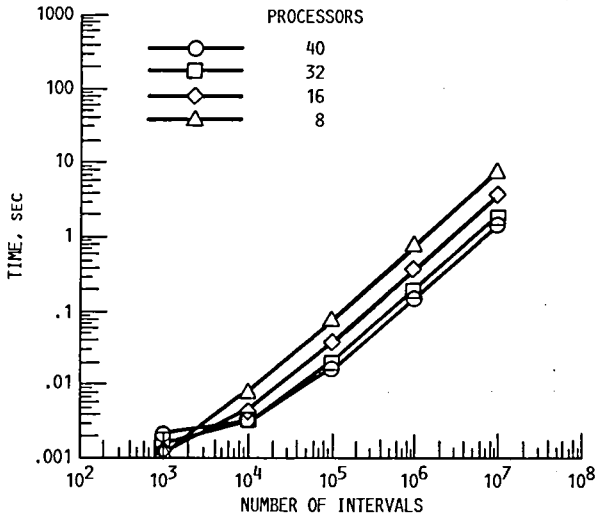

FIGURE 3. - PI PROGRAM USING PRIORITIZED COMPUTATION, 32-BIT MATH, AND T800 FLOATING-POINT PROCESSORS.

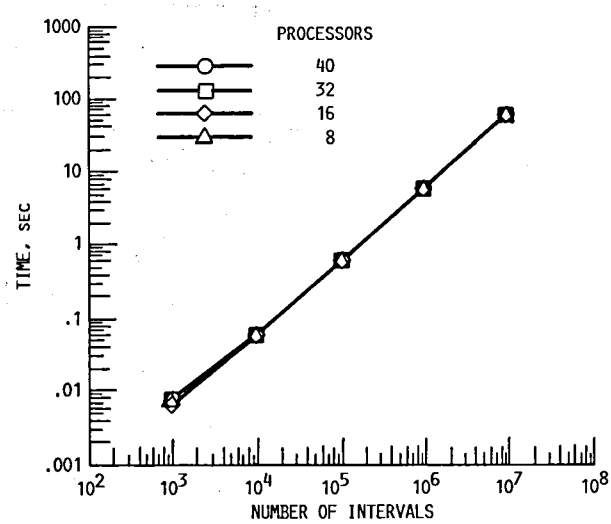

FIGURE 4. - PI PROGRAM USING PRIORITIZED COMPUTATION, 32-BIT MATH, AND T800 FLOATING-POINT PROCESSORS.

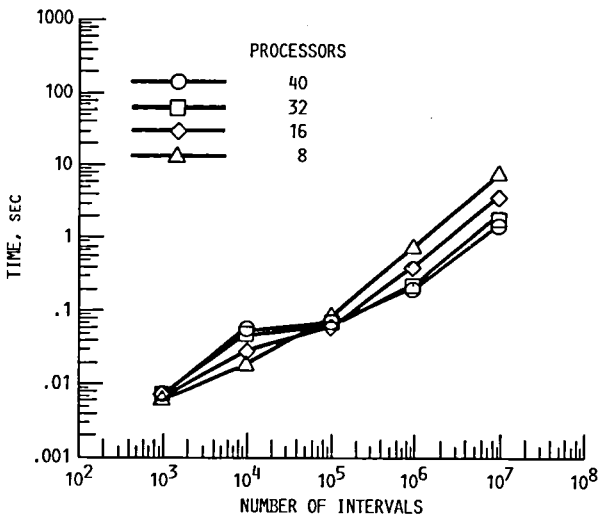

FIGURE 5. - PI PROGRAM USING ALL LOW-PRIORITY PROCESSES, 32-BIT MATH, AND T800 FLOATING-POINT PROCESSORS.

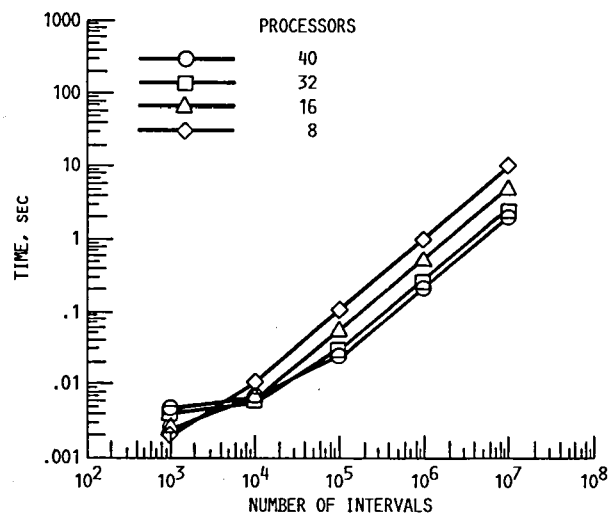

FIGURE 6. - PI PROGRAM USING PRIORITIZED COMMUNICATION, 64-BIT MATH, AND T800 FLOATING-POINT PROCESSORS.

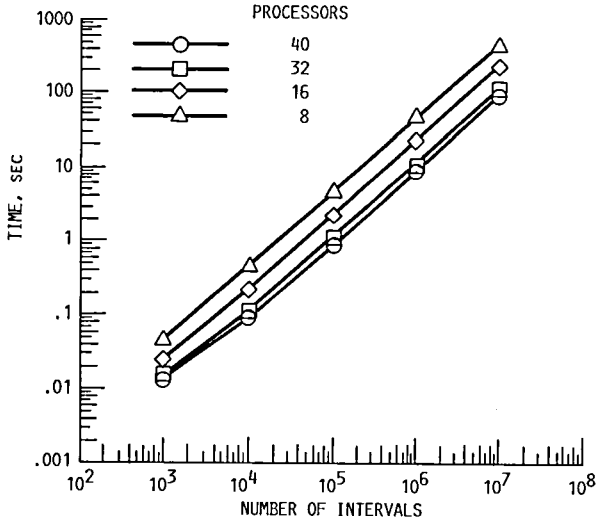

FIGURE 7. - PI PROGRAM USING PRIORITIZED COMMUNICATION, 64-BIT MATH, AND T414 INTEGER PROCESSORS.

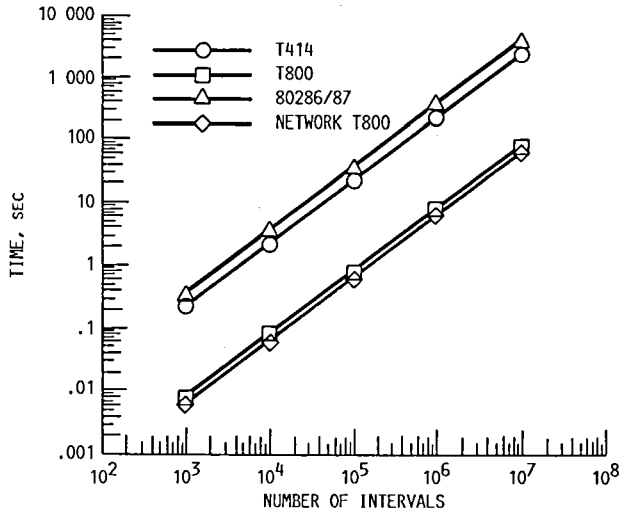

FIGURE 8. - SINGLE PROCESSOR PERFORMANCE SOLVING PI PROGRAM USING 32-BIT MATH.

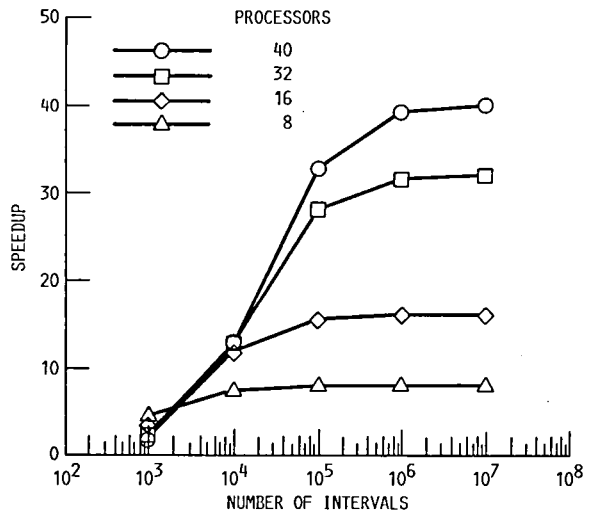

FIGURE 9. - PI PROGRAM SPEEDUP USING T800 FLOATING-POINT PROCESSORS.

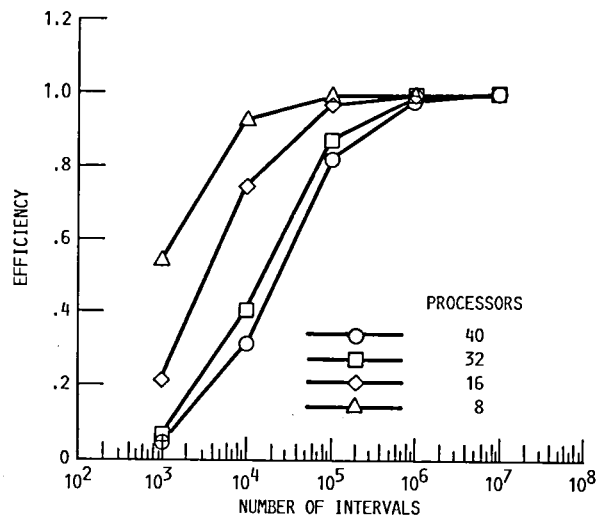

FIGURE 10. - PI PROGRAM NETWORK SOLUTION EFFICIENCY USING T800 FLOATING-POINT PROCESSORS.

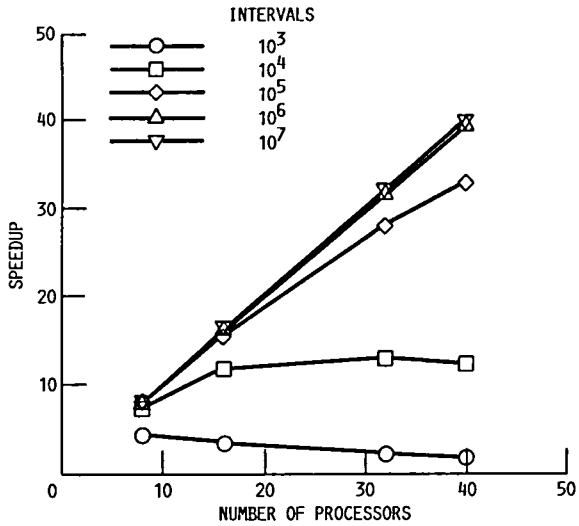

FIGURE 11. - SPEEDUP FOR PI PROGRAM.

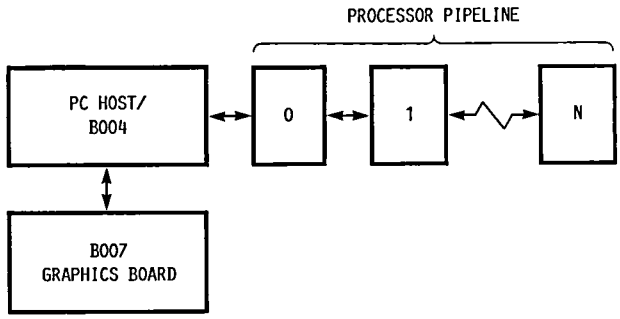

FIGURE 12. - NETWORK USED FOR ROOT VISUALIZATION PROBLEM.

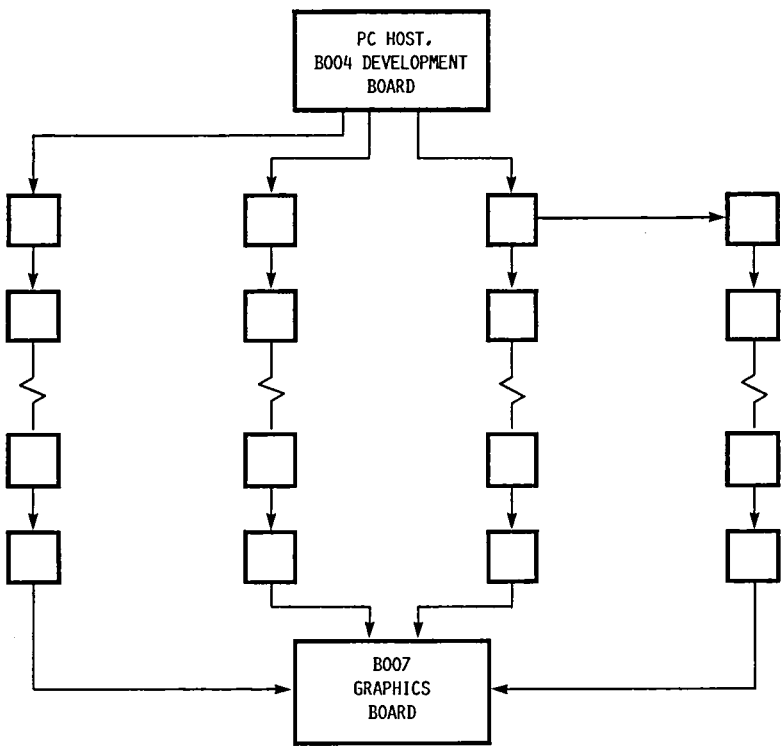

FIGURE 13. - SUGGESTED HIGH-BANDWIDTH NETWORK TO TAKE ADVANTAGE OF AS MANY TRANSPUTER LINKS AS POSSIBLE.

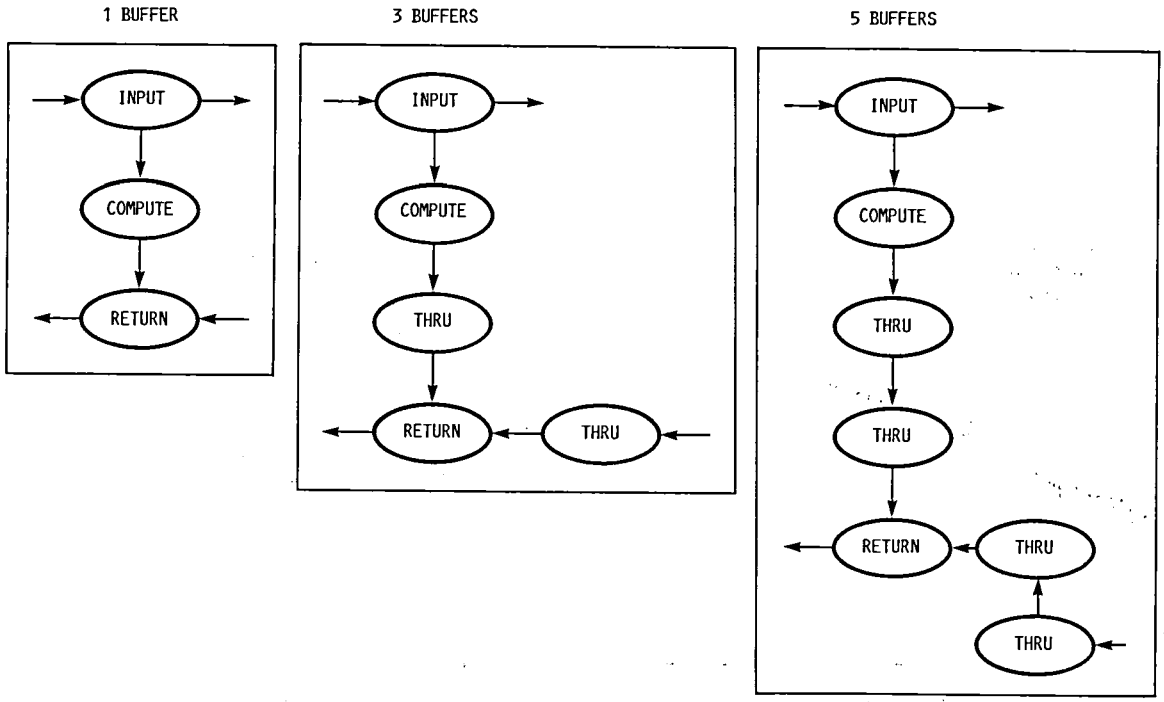

FIGURE 14. - OUTPUT BUFFER CONFIGURATIONS FOR ROOT VISUALIZATION TESTS.

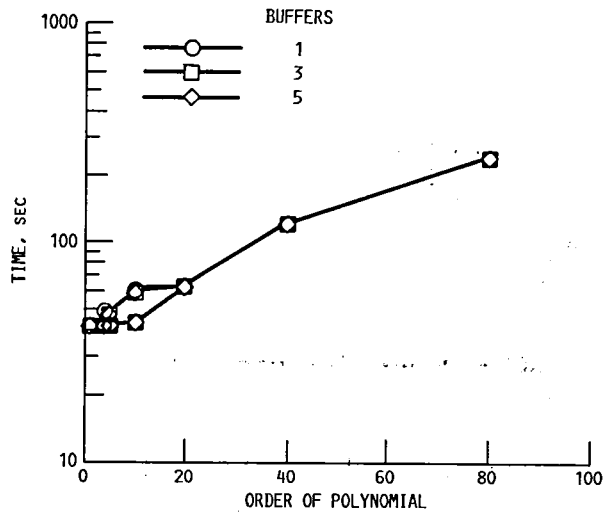

FIGURE 15. - ROOT VISUALIZATION TEST PERFORMANCE USING 32-BIT TRANSFER PROTOCOL.

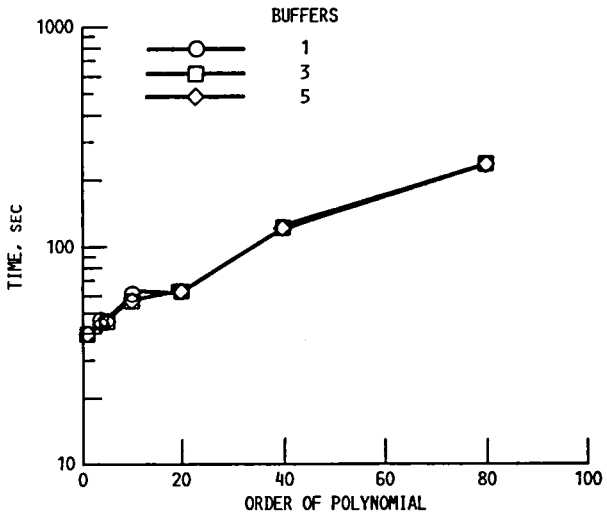

FIGURE 16. - ROOT VISUALIZATION TEST PERFORMANCE USING 16-BIT TRANSFER PROTOCOL.

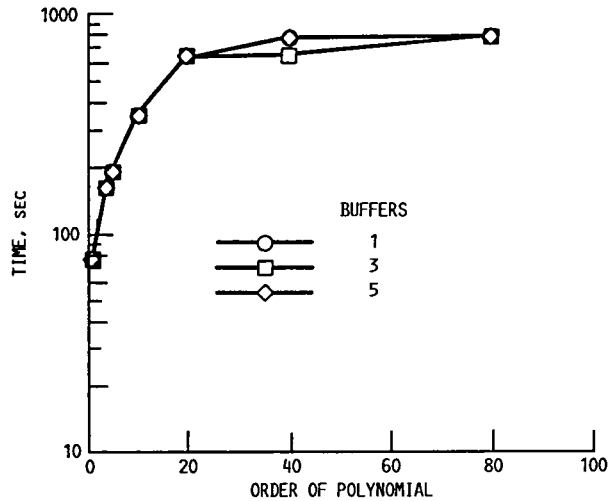

FIGURE 17. - ROOT VISUALIZATION TEST PERFORMANCE USING RUN-LENGTH (RL) ENCODING.

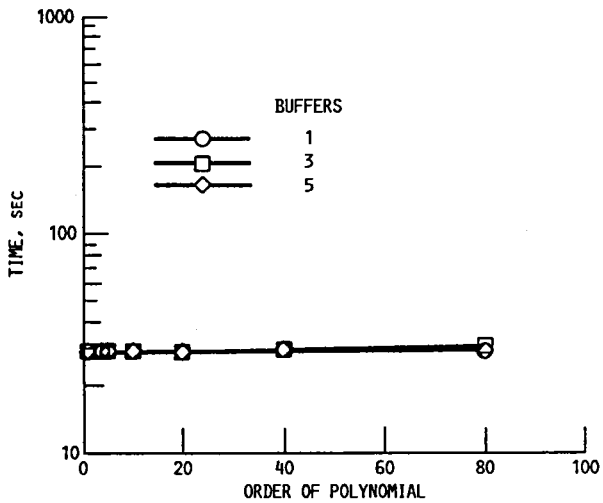

FIGURE 18. - ROOT VISUALIZATION TEST PERFORMANCE USING 32-BIT TRANSFER PROTOCOL AND T800 PROCESSORS.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA TM-101297 ICOMP-88-14		

4. Title and Subtitle	5. Report Date
Implementing Direct, Spatially Isolated Problems on Transputer Networks	August 1988
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
Graham K. Ellis	E-4278
	10. Work Unit No.
	505-63-1B

9. Performing Organization Name and Address	11. Contract or Grant No.
National Aeronautics and Space Administration Lewis Research Center Cleveland, Ohio 44135-3191	
	13. Type of Report and Period Covered
12. Sponsoring Agency Name and Address	Technical Memorandum
National Aeronautics and Space Administration Washington, D.C. 20546-0001	14. Sponsoring Agency Code

15. Supplementary Notes

Graham K. Ellis, Senior Research Associate at the Institute for Computational Mechanics in Propulsion, NASA Lewis Research Center (work funded under Space Act Agreement C99066G).

16. Abstract

Parametric studies have been performed on transputer networks of up to 40 processors to determine how to implement and maximize the performance of the solution of problems where no processor-to-processor data transfer is required for the problem solution (spatially isolated). Two types of problems were investigated in this study. A computationally intensive problem where the solution required the transmission of 160 bytes of data through the parallel network, and a communication intensive example that required the transmission of 3 Mbytes of data through the network. This data consists of solutions being sent back to the host processor and not intermediate results for another processor to work on. Studies were performed on both integer and floating-point transputers. The floating-point transputer features an on-chip floating-point math unit and offers approximately an order of magnitude performance increase over the integer transputer on real valued computations. The results indicate that a minimum amount of work is required on each node per communication to achieve high network speedups (efficiencies). The floating-point processor requires approximately an order of magnitude more work per communication than the integer processor because of the floating-point unit's increased computing capability.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Parallel processing Transputer Performance calculation	Unclassified - Unlimited Subject Category 61

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No of pages	22. Price*
Unclassified	Unclassified	58	A04

NASA FORM 1626 OCT 86

*For sale by the National Technical Information Service, Springfield, Virginia 22161

National Aeronautics and
Space Administration

Lewis Research Center
ICOMP (M.S. 5-3)
Cleveland, Ohio 44135

Official Business
Penalty for Private Use $300

FOURTH CLASS MAIL

ADDRESS CORRECTION REQUESTED

Postage and Fees Paid
National Aeronautics and
Space Administration
NASA-451

NASA