A Guide to Parallel Programming

Using

PCS-Linda and the Parallel Lan System

Parallel Computer Solutions

November 30, 1991

Contents

Linda

Linda is a parallel processing coordination language. To most computer scientists and engineers, this is a new concept. Commonly, sequential languages such as C and Pascal are enhanced to support parallel processing; Concurrent C and Parallel Pascal are two such languages. However, if we look at the actual code that makes up a parallel application we see that not all of the code is parallel. It can't be. If a parallel application was completely parallel, one instruction would be executed on each machine. This is not reasonable of course. Parallel programs are written such that certain pieces of code are farmed off to other processors while other code is executed on the host machine to pull all of the pieces together. Now there are machines where this is not true but we will not discuss those. These pieces of code farmed to other processor are typically small and perform a specific operation on some data. They are commonly called processes.   Let's look at a common real world situation to put these ideas together.

Large, busy offices in a corporation usually are made up of many different people doing some task or process. If we were to let this office run itself, it would become a mess in a very short time. Some people would be doing tasks that had already been done. There would be miscommunication. You can probably think of many more things that could go wrong. What this office needs is an office manager. An office manager isn is responsible for pulling the individual people into a working team. Each team member working for the greater good of the corporation.

The Language Linda

The same can be said for Linda. Linda is a language developed at YALE University and is copyrighted by Scientific Computing Associates, Inc. Linda however lacks the common features you and I commonly think of when we hear about a new language. There are no loops, decisions, records, etc in Linda. All of these common functions are provided in some other language used in conjunction with Linda. This language could be C, Pascal, Forth or any other language. The primitive functions of Linda are coded in a particular language and can be used just as any other statement can be used.

Linda coordinates the activities of many different executing processes on different machines. As we will see later, these machines do not have to be a single parallel machine such as the Connection Machine. The Linda system we will be using coordinates the activities of different processes running on IBM PCs connected by an ethernet local area network.

As we describe some of the characteristics and feature of Linda, we will introduce our version of Linda called PCS-Linda. There are features of Linda that can be difficult to implement on top of an existing language and compiler therefore, we have not implemented everything in our version.

Linda Primitives

It may be surprising to find out that Linda has just six instructions. These six instructions are in, out, rd, eval, inp, rdp. Each of which will be discussed in detail later in this section. First we must talk about how Linda coordinates processes. Linda uses a distributed data structure called a tuple. A tuple is a data structure that can be accessed by any process thus the distributed part.

Tuples

Tuples have two parts: a name and some number of elements. The name of a tuple is any combination of up to sixteen
characters. The name serves as the primary matching characteristic for the tuples.

(name, e1, e2, e3, e4, e5, e6)

The name can be either a variable (string) or enclosed in single quotes. The elements are filled from left to right,

Elements

A tuple can have from zero to six elements. Each of the elements will be a value or variable of a particular type
supported in the host language Linda is coded in. In PCS-Linda the types can be integer, real, array, and record. Strings are not allowed in the current PCS-Linda system. Elements, in addition to having a specific type, can have a further description of actual or formal.

Actuals

When we speak of actual, we are concerned with an actual value. This value can be either the numerical representation or can be contained in a variable. When using a number as an element, we simply state the number in one of the element fields.

```
( 'example1', 1, 2 )
```

is an example of a tuple with name 'example1' and actuals 1 and 2, each of type integer. If we had two variables a and b of type integer,

```
a, b : integer;
```

and we wanted to use the values contained in these variables, we would precede each of the variable by the & symbol. The tuple

```
( 'example2', &a, &b )
```

would be an example of a tuple with name 'example2' and actuals a and b, of type integer. If this tuple was used in a program, the integer values assigned to the variables a and b would be compiled into the tuple definition.

In addition to integers, reals or floating point values can be used as actuals. Thus we can have

```
( 'example3', 3.141592, 1.2e+08 )
```

can also be used as a tuple. In most cases, the only thing that differentiates an integer from a real number is the decimal point and therefore must be included when specifying a real number.

Formals

Formal elements are more like variables. They are used to collect a value from a tuple. They are distinguishable by the lack of the & symbol. A tuple with formal elements would appear as

```
( 'example4', a, b, c )
```

The variables a, b, and c would be assigned to values returned after a match has taken place on the tuple. Formal variables can be any of the previous data types mention except strings.

Tuple Space

Now that we have tuples, we have to have a place to keep them. This storage place is called the tuple space. The tuple space could be envisioned as a large bag full of tuples.

The tuple space can be likened to shared memory in that all processors have access to the tuples in the tuple space. However, unlike shared memory, there are no critical sections in the tuple space. Any of the tuples can be used by any processor at any time. Now there are ways to control access to the tuple space by using tuples that emulate semaphores and the like. In order for a process to access a tuple, a matching has to take place. Thus a processor would sent a template of the tuple it would like to match in the tuple space. The machine holding the tuple space would perform a search on all of the tuple in the tuple space. As soon as a match is found, the matching tuple is sent to the processor that requested the match.

The tuple space in our system is located on a single machine called a master. The master is responsible for holding tuples and matching tuples. When tuples are put into the tuple space, the master does not check for duplicate tuples. Any number of
duplicate tuples can exist in the tuple space. There are specific rules that the master will follow when matching tuples.

Tuple Matching

RULE 1 : Tuple names must match both length and character for character.

RULE 2 : Actuals match actuals if of the same type and contain the same value.

RULE 3 : Actuals match formals if they are of the same type and length.

RULE 4 : Formals never match formals.

Let's look at each rule using some examples. Assume we have the following tuples in our tuple space.

```
   ( 'stuff', row, col, 1 );

     ( 'coord', &x, &y );

 ( 'work', &segments, &adder );

     ( 'Junk', 1, 3, 5 );
```

Row, col, x, y are integers;
Adder is a procedure;

Segments is a record of length 18;

We have to match the tuple ('stuff', 100, 200, start) with a tuple in our tuple space. First off, RULE 1 is satisfied with the first tuple in tuple space because they both have names of length 5 and match character for character (stuff = stuff). The elements of the tuples are matched next. The tuple to be match has an actual element of type integer in the first position and value 100. The first tuple in tuple space has a formal of type integer in the first position. By RULE 3, the first elements match in each of these tuples. Further study shows that the second and third elements match as well.

Further Examples

The tuple ('stuff', 3.4, 200, start) does not match the tuple ('stuff', row, col, 1) because the types of the first element are different.

The tuple ('stuff', &a, &b, &start) may or may not match the tuple ('stuff', row, col, 1). A determination cannot be made because we do not have a value for the variable start. If start equals one, then we have a match but if start equals any other integer, the tuples do not match.

Note on real elements

As we all know, computers have a tough time representing real or floating point numbers well. Some numbers can be represented exactly such as 0.5 but how does a computer represent the value of 2/3. At some point, the computer will have to round to 0.666666667 which is not correct. This inaccuracy must be kept in mind when using real number as element types in tuples. If during a calculation, a real number is used in a tuple, there may not be a match because of rounding. Thus it is probably wise to avoid matching on reals if at all possible. In most cases, you will probably get a match but that one time when you don't, keep the above in mind when looking for the problem.

Linda Instructions

Now that we have tuples and a place to put them, we must look at the individual Linda instructions and determine what each one of them does and how to use them. This section will be ended with the common 'Hello World' problem coded in Linda and Pascal. The first four Linda instructions discussed are the most common ones used. The last two are variants of two of the common ones.

IN

The IN instruction is used to request a match on a tuple from tuple space. The format of the instruction is

```
IN ( name, e1, e2, e3, e4, e5, e6 )
```

Only the elements necessary are included in the tuple. The tuple specified with the IN instruction is sent to the master for a match with a tuple in tuple space. If there is a positive match, the master will return the matching tuple. If there are any formal elements, they are assigned the appropriate values from the matching tuple. If there is not a tuple in tuple space that matches the tuple sent by the IN instruction, the master keeps the tuple and continually check the tuple space for a match. In the meantime, the processor that sent the IN instruction will block execution until a matching tuple is sent. Once that master has a match, it is sent. When the master sends the matching tuple to the requesting processor, it is permanently deleted from tuple space. If there are multiple copies of the same tuple in tuple space, the first one is taken. The master will take the first tuple that it determines matches the tuple sent with the
instruction.

RD

The RD instruction is somewhat identical to the IN
instruction. The format of the RD is

```
RD ( name, e1, e2, e3, e4, e5, e6 )
```

The RD instruction sends a tuple to the master for a match with a tuple in tuple space. If the master has a match, it sends a copy of the tuple found in tuple space. If the master does not have a match, it will continually search tuple space for a match. Meanwhile, the processor that sent the RD instruction will block execution until a tuple is returned. When the master finds a tuple, it sends a copy of the tuple to the requesting processor. The tuple is not deleted from tuple space. This allows a single tuple to be read from any processor without the overhead of INing the tuple and OUTing the tuple just to get a copy of the tuple. The RD instruction should be used when communicating common data to a number of different processors.

OUT

We have seen how to request tuples from tuple space but how do we get tuples into tuple space to begin with. The OUT instruction enables us to put tuples into tuple space. The format of the OUT instruction is

```
out ( name, e1, e2, e3, e4, e5, e6 )
```

The tuple with the OUT instruction is sent to the master and is put into tuple space. Any number of the same tuples can be put into tuple space. In all cases of actuals, the actual values are sent to the master in the tuple. There are no pointers referencing data in a different processors memory.

The master does not respond to the OUT instruction. It is simply taken for granted the tuple is put into tuple space. Once an OUT has been performed, any processor can access the tuple including the processor that issued the OUT instruction in the first place.

EVAL

The EVAL instruction is the most important of the Linda instruction. Without it, the other instructions are of no use. The EVAL instruction is used to put code into tuple space. This code is picked up by workers and executed. The format of the EVAL instruction is

```
eval ( name, e1, e2, e3, e4, e5, e6 )
```

In PCS-Linda, the eval instruction has not been fully developed to the specification of the original Linda EVAL. In PCS-Linda the format of the EVAL instruction is

```
eval ( 'work', &procedure );
```

All EVAL instruction must name the tuple 'work'. Because tuple with the same name can reside in tuple space at the same time this is not a problem. The first element of the tuple will be an actual designated by the & operator and the name of the
procedure that is to be put into tuple space. So if we wanted a procedure

called MANDEL to be put into tuple space to be executed, the following instruction would be used

```
eval ( 'work', &mandel );
```

INP

The INP instruction is identical to the IN instruction except for one condition. If the master finds a tuple match when the instruction is received, it will return the tuple. If a tuple is not found in tuple space, the master returns a null tuple to the requestor. Thereby allowing the INP instruction to be evaluated as either true or false depending on whether or not a tuple was matched from tuple space. PCS-Linda implementation information of this instruction will be given later.

RDP

The RDP instruction is identical to the RD instruction except the master does not continually search for a match if its first attempt in unsuccessful. If a tuple is found, the matching tuple is returned to the requestor. If the master does not find a matching tuple in tuple space, the master will return null. Thus allowing the RDP instruction to be evaluated as either true of false depending on whether or not a tuple is returned.

Those are the six instructions that make up the Linda coordination language. Let's look at an example of a 'Hello World' program using Linda and Pascal.

Hello World Example

We will assume we are working on a system with 8 worker processors. The beginning of our program would be standard Pascal.

```
program hello;

const
  num_proc = 8;
```

Next we have to write the procedure for the workers.

```
procedure world;
var
  count : integer;

begin
```

```
                  in ( 'count', count )
                  inc ( count );
                  out ( 'count', &count
                  out ( 'hello', &count );

                end;
```

Now the main procedure.

```
                begin

                  out ( 'count', 0 );
                  for i := 1 to num_proc do
                    eval ( 'work', &world );

                  for i := 1 to num_proc do
                    begin
                      in ( 'hello', proc );
                      writeln ( 'Hello from processor', proc );                         end;
                  in ( 'count', 8 );

                end;
```

The program begins with the tuple called COUNT being put into tuple space with an integer actual of 0. This is followed by eight copies of the WORLD procedure; one for each worker. The main program goes into a loop requesting a HELLO tuple one at a time. Not that the first element is a formal, thus we are only matching on the name of the tuple HELLO. The formal element will have some value on each loop iteration. The numbers my not be in order. As each tuple is found in tuple space, a message is printed that says Hello from processor -. The code ends with an IN which cleans up the tuple space.

The workers are instructed to IN the COUNT tuple, increment the number it finds in the first element position and put the tuple back in tuple space for the next processor. After it does this, it puts a tuple in tuple space called HELLO with the number it put into the COUNT tuple. Each worker will get a different number to report back to the main code.

Now as we stated above, this code will receive eight tuple with the name HELLO is any order. If we changed the main programs loop slightly, we could guarantee to get the HELLO tuple in order from 1 to 8.

```
                for i := 1 to 8 do
                  begin
```

```
                    in ( 'hello', &i );
                    writeln ( 'Hello from process - ', i );                    end;
```

We have changed the first element from a formal to an actual. Thus instead of the master having to match the name, it must match the first element as well. Therefore, the it will return a HELLO tuple only when the first element is a 1.

Parallel Lan System

Now that we know how to program Linda and we have seen a parallel program, we need to begin doing some real work. So now we can sit down to our PC and begin programming the examples.

Well not exactly. Our common PC is a single processor machine. We have no way of dividing up work and allowing different processors to work on the pieces. We have two options. We can purchase an expensive parallel machine for several thousands of dollars. Have it installed and teach ourselves the things necessary to program the machine. Or we can use the Parallel Lan System.

The Parallel Lan System is a software package that allows IBM PC or compatible machines to operate as a parallel processor. The machines must be connected together by an ethernet local area network. A minimum system consists of three machines: a master, worker, and a developer.

Using Turbo Pascal and the Parallel Lan System (PLS), a parallel program can be constructed for any parallel algorithm we so desire. In addition, we have a version of Linda called PCS-Linda that we will use to coordinate the activities of the different processes created our the parallel program.

The PLS was created to co-exist with other network products on the market such as Novell Netware and DECNet. The system will not interfere with NCSA or PCSA or any of the TCP/IP programs. The Parallel Lan System can even be configured to run several parallel programs on the same network sharing the same processor of the system.

Configuration

The configuration of the Parallel Lan System is very important to the efficiency of the parallel system. The remaining sections will document the different components of the PLS. Several examples will be presented in the end of the guide.

Requirements

The Parallel Lan System operates by using an ethernet local area network (LAN). LANs are very popular among educational instructions and businesses. Various packages are available to run on networks including

Novell Netware and MS Lan Manager.

The Parallel Lan System communicates on an ethernet by way of a packet driver. Most ethernet card manufacturers have these drivers available at no cost. In addition, there are public domain packet drivers available for most cards.

Packet Drivers

Packet drivers are terminate and stay-resident (TSR) programs which act as an interface between a developer and an ethernet card. Ethernet cards use a hardware interrupt of a PC. When this interrupt is activated, the packet driver code is activated to perform some function. Likewise, the Parallel Lan System software is able to activate a software interrupt which also activates the packet driver code for its own use. Software can be written that locates the packet driver installed in a machine thus allowing an PC and ethernet card to be used without changing the system software.

Manufactures who provide packet drivers will also include information on how to install them. Packet drivers can be used with most network packages. An advantage of using packet driver is multiple applications can access the same ethernet card. The packet driver has the ability to give a packet from the LAN to one application or another based on a type field located in the packet.

What we want to do now is install the packet drivers for the machines the system will run on. Follow the instructions given by the manufacturer. With that, we want to verify that everything to this point is operating correctly. On the disk labeled system software is a file called PKTINFO.EXE. When executed, this program will give us information about the ethernet card and packet driver installed on a particular machine. After the packet driver has been installed in a machine, place the system disk in driver A: and type

```
A:PKTINFO
```

If a page of information appears on the screen, the packet driver is working correctly. If a message appears saying no packet driver was found then the packet driver was not installed
correctly.

After the packet drivers have been verified we want to check the LAN itself. Located on the same disk is a program called TRANS.EXE. This is a simple LAN communication program that will send and receive packets from on PC to another. Pick two machines to execute this software on. One one of them, write down the ethernet address as shown by PKTINFO. Inset the system disk in drive A: and type

```
                A:TRANS
```

Take the disk out and do the same for the other machine. On the PC that will receive the packets press SHIFT and R. We now want to select what receive mode to use. Press 2. The screen will blank and a status bar will appear at the top. On the sender machine press SHIFT and S. We need to select a receive mode as we did for the receive machine. The reason is because the receiving machine will send an acknowledgement packet back to the sender. So press 2. We are now asked if we want to use the broadcast feature. Press N. The program is asked for the address of the receive machine. Enter the six bytes written down separated by spaces and press return. Enter a message to be sent to the receiving machine and press enter.

The screen will blank and ask you to press a key to send a packet or shift Q to quit. Press any key. If you look at the receiving machine, you message should be on the screen. On each of the screens, there should be a 1 in the upper right hand corner. This indicates that one packet has been sent and one packet has been received. You can continue to press any key to send additional packets. Press SHIFT and Q when ready to quit.

Master

The master of the Parallel Lan System is the most important machine. It can be likened to the server of a local area network. It must be powerful and able to handle a flood of activity by the worker nodes and the developer. As documented in the manual Using the Parallel Lan System, the master should consists of

* At least a 25-MHZ 386 IBM PC or Compatible.

* 4 MB of RAM

* 16-bit 32k buffer Ethernet card

Anything above these specification will allow for better efficiency in the system. Other than running the system software for the master, there is nothing that can be done to the master.

Duties

The duties of the master are as follows

* Administer the MAIN tuple space of the PLS.

* Receive and interpret Linda instructions from multiple processors.

* Keep track of unanswered Linda instructions (IN,RD).

Those are the only responsibilities of the master. However it is a larger responsibility than it may appear.

Multiple Processor Communication

When considering the concept of multiple processor communication, image this situation.   You are sitting in the middle of a ring of sixteen people. All of these people are trying to hold a conversation with you AT THE SAME TIME. The human mind simply cannot handle that much information at the same time. Most people can't even keep track of a single conversation. That is one of the jobs of the master.

Things begin with the ethernet card receiving a packet. This packet is put into some memory set aside for incoming packets. The master system software periodically checks to see if there are any packets in the first incoming buffer. If there are packets, it will take as many as there are and partial processes them and put them into a secondary buffer. This step is crucial. The first buffer is static. In other words, it is of a constant size. As much information about this buffer as possible was defined when the master program began executing. If it were to overflow, some packets would be lost (the sender would send duplicates however. But at a lose of execution time). After the packets are in the secondary buffer, the system software will processed them fully one at a time as it has time. Processed packets will end up in one of two places. Packets which represent the Linda instructions OUT and EVAL windup in the TUPLE SPACE. Packets representing IN, INP, RD, and RDP will end up in a request queue. The master system software picks from the request queue when it tries to match tuples.

In addition to the above queues, the master also has internal data structures that it uses to guarantee that no duplicates packets are processed. We all know what its like to be told the same thing over and over. The master doesn't like it either therefore it eliminates duplicates. Also, the master must keep track of the order of packets from different processors. Just like we try to say a persons first name before their last name, the master must guarantee that a packet sent before another packet arrives first.

For more information on the about technical data about the system software, refer to the document The Parallel Lan System -A Look Inside.

Different Masters

There is a difference in performance between different masters. We are going to take a look at how masters can influence the overall speed of a parallel program. The masters used are

* a 4.77-MHZ 8086 IBM PC Compatible, and

* a 25-MHZ 80386 IBM PC Compatible.

Each of masters was used in a comprehensive set of 40 tests using the Linda mandelbrot program explain in a later chapter. The test documented here was performed using from 1 to 16 workers (4.77 MHZ 8086 PC's using 8087 math coprocessors) each doing 4 columns per computation. The following table shows the difference between the masters.

Cpus	8086 Master	80386 Master	Speedup
1	642.52	577.91	11%

2	358.89	274.89	24%
4	210.80	141.65	33%
8	188.72	80.36	58%
16	----.--	88.87	-----

The chart shows the total seconds for each test using the 8086 master and the 80836 master. The far right column shows the speedup between the two systems. There is a considerable speedup as we approach 8 workers. There was no test performed for 16 workers using the 8086 master.

Notice the increase in time between 8 and 16 workers. This example also illustrates the idea that adding more workers does not necessarily decrease execution time of a program. A faster master does indeed make a considerable impact on the speed of the execution of the program.

Machine

The faster the machine, the faster the tuple space can be searched for needed tuples. The most work the master will perform is interpreting tuples sent to it. The faster it can do this the better.

DOS

Microsoft recently released DOS 5.0. This DOS has many memory saving features that the system can benefit from. IF the master is equipped with 1 MB of memory or more, DOS can be loaded into high memory as well as device drivers and TSR's. All of this can save precious lower memory.

The master software uses lower memory (as well as extended) to hold the tuple space. A system using DOS 3.3 or 4.01 will have approximately 300k of heap space available for the tuple space. Dos 5.0 increases this amount to approximately 360k.

Extended Memory

If we have 1 MB available we can take advantage of the features of DOS 5.0 Any memory over 1 MB can be configured as extended memory by using an extended memory manager. The master software will take advantage of any extended memory available.

Using extended memory for some of the tuple space is expensive as far as processing speed is considered. The software uses a system put into public domain to manage extended memory. Unlike conventional memory below

640k, extended memory cannot be accessed directly. A system of segments is created in extended memory. These segments are copied into conventional memory to be
manipulated and then put back. The segments are 32k in size. This means there is a 64k memory copy performed for each and every packet that must be put into extended memory. Now simple memory management has been put into place which will keep the most recently used extended memory extent in conventional memory. However, to execute programs which has a large amount of data, extended memory is essential. At Parallel Computer Solutions, a 4 MB system has been used in all cases with no problem as far as space requirements.

Network Cards

Not all cards are created equal. We have 8-bit, 16-bit, and 32-bit cards available for PCs.   To the best of my knowledge, 32-bit cards are available for EISA bus system only. Therefore, most PC's will use either the 8 or 16 bit cards. If we have a 80286 or better machine, we will want to use 16-bit cards. 16-bits ethernet cards have several advantages.

The first is the addition of 8 data lines. More information can be transferred on 16 lines than 8 lines. A second reason is buffer size. Most 8-bit ethernet cards have an 8k buffer for incoming packets. For most applications this is fine but the master will be receiving packets from many different PCs at the same time. We want to have a large buffer available if the master is busy searching tuple space or some other system function. 16-bit ethernet cards have either 32k or 64k buffers for incoming packets.

You usually get what you pay for when buying a computer system. The master is the most important component in the Parallel Lan System and therefore it should be the best system available.

Master Installation

Obtain a blank diskette to copy the master software from the system diskette. To make the system easy to install, create a bootable diskette with the appropriate memory managers, etc for you system. Copy the packet driver for the ethernet card onto the new diskette. Copy the file MASTER.EXE to the net diskette. Label this disk as bootable and containing the master system software.

Execute the master software on the appropriate machine by typing

A:MASTER

A message will appear telling how much extended memory is available and used by the master system. Press return to get to the next screen. The following should appear

(master screen - initial picture)

This is the initial screen for the master. The master sits in a loop waiting for a tuple to be sent to it. Under normal circumstances, the master does not need any attention. The operation of the master can be disabled by pressing any key. As activity starts on the system, the screen will change, this screen

(master screen - 2 cpu picture)

shows that 2 cpus are active on the system and the packets sent and received from those cpus. In addition, the sizes of the heap and extended memory is given.

Worker

The workers are an important part of the Parallel Lan System. The workers do one thing: perform calculations. The least powerful machine available for a worker is:

* 4.77 Mhz 8088 IBM PC or Compatible

* DOS 3.3

* 640K RAM

* 8-bit ethernet card

Just as in the case of the master, the performance of the Parallel Lan System can be determined by the power of the worker machines. Testing of the Parallel Lan System was performed on both 4.77 Mhz 8088 workers and 20 Mhz 80386 workers. Both systems provided speedup simply because of the parallel execution of the test program. But the 80386 workers gave additional speed advantages by as much as 60/80%. The following four charts show the times required to perform mandelbrot using 8088 workers and 80386 workers. The master in the first two charts was an 8088 machine. In the second two charts, the master was an 80386 machine.

(charts)

In an established LAN we do not have much choice in what type machine is used as a worker. The charts should show you that processor power certainly makes a difference in both the master and workers.

Worker Installation

We must now install software on each of the worker machine. Follow the previous directions for setting up a master diskette but instead of copying master.exe we need to copy a worker program.

Turbo Pascal is available in two different versions: 5.5 and 6.0. They are different in code generation. Therefore, we have two different worker pieces of software called worker5.exe and worker6.exe. If you are using Turbo Pascal 5.5 then copy
worker5.exe to a:worker.exe and if you are using Turbo Pascal 6.0 then copy worker6.exe to a:worker.exe.

Insert the diskette into driver A: and type

```
A:worker
```

The screen will blank and you will be asked the address of the master. If you did not write the address of the master down, you can look on the screen of the machine it is executing on. In the upper left hand corner of the screen is the ethernet address of this machine. On each of the worker systems, enter the six bytes separated by spaces. Once the system has digested the master address, it will attempt to establish communications with the master.

Recall that the purpose of the worker is to execute code given to it by the developer system. The worker will get work from the master by sending a tuple of the form

```
in ( 'work' );
```

to the master. All packets sent over the ethernet are given a specific number in order to keep a sequence. the system software gives the number 2 to the first packet sent out by any system. We can verify that a worker is communicating with the master by the messages put on the master and worker screens. A worker has communicated successfully if on the screen of the master is a message IN - 2 for each of the workers.

Since a packet has been sent to the master, the master has to acknowledge the packet, therefore each of the workers should have a message in the right most box with the number 2 next to it. If this is the case, then this worker has been successful added to the system. If this is not the case, then either the master's address was not correctly entered or the address in not that of the master. The worker software halts when a key is pressed. So if no message appears on the screen , press a key on the worker machine and try again by executing the worker software again.

Developer

The developer is a machine on the system which runs and coordinates the executing parallel program. It will begin a program by submitting any number of worker processes. The responsibilities of the developer are

* Submit jobs for worker processors

* Coordinate the submitted jobs

* Produce results

The developer is the foreman for the Parallel Lan System. A programmer will code a parallel program on the developer, compile it, and run it without moving to different machines. One way to look at it is you are programming a single PC which just happens to have any number of subprocessors available to help speed up a particular program.

Again it needs to be pointer out that if you are using Turbo Pascal 5.5, the workers should be executing worker5.exe and if you are using Turbo Pascal 6.0, the workers should be executing worker6.exe.

The Parallel Lan System developer software is setup to initially recognize files with an extension of .PLS. Turbo Pascal and our system files can be set up in such a way as to simplify use of the system. In the root directory of you hard drive create a directory called PLS using the command

```
mkdir pls
```

Move into that directory with

```
cd pls
```

Insert the system software diskette into drive A: and copy the following files into our new directory

```
work*.tpu

both*.tpu

*.pls

linda.exe
```

```
*.doc
```

Now back out of this directory with

```
cd ..
```

Using an editor change your autoexec.bat file to include the turbo directory in the system path. If no PATH directive appears in you autoexec.bat file, enter the following line

```
path c:\tp - or whatever the turbo pascal directory name is
```

After you have changed the file type

```
autoexec
```

This is make the change effective. Now change into your PLS directory and type

```
turbo linear.pls
```

Turbo's main screen will appear. We need to make sure the compiler settings are set correctly before we do anything else. Press ALT and O, move to the COMPILER and press return. Move the bar to FORCE FAR CALLS and change the entry to YES. Check that the entry BOOLEAN EXPRESSIONS is set to SHORT-CIRCUIT. Move to the first menu and select the SAVE OPTIONS entry and press return. Save the new options.

Now on the screen will be the parallel program for solving linear equations. Try to compile this program by pressing F9. You should get an error on the first Linda command IN.   What we need to do is execute the conversion program LINDA.EXE. Linda.exe is explained in the document, Linda Conversion Program User Guide. To do this in a convenient manner, press ALT and F. Move to the entry EXIT TO DOS and press return. This will exit us to DOS but keep Turbo Pascal in memory. Run the conversion program by typing

```
linda linear
```

The linda program will create a file called Linear.pas that is made up of Turbo Pascal statements only. Type exit and press return. This will return us back to Turbo Pascal. We are still working on the file Linear.PLS. Press ALT and F and move the bar to PICK and press return. Pick a new file and enter linear.pas. This will bring up the file linear.pas. Press F9 to compile the program, it will compile successfully. If there had been an error, we would have wanted to press ALT and F, move the bar to PICK and select linear.pls to make any changes, run linda again, and select linear.pas. Although this may seem a hassle, it is the only way to incorporate a preprocessor in the Turbo environment. Future releases will not require this.

Compiler Directives

During the development of the system, several things were learned about how Turbo Pascal (TP)   generates code. TP allows for several different types of float point variable. Reals, double, extended and several other are the chooses that we have.

In most applications, real variables are good enough for our calculations. TP will generate code for reals automatically. In order to generate code to handle double or extended variables, the compiler directives {$N+,E+,F+} are necessary.

So what the problem. The problem comes about when there are constants in your code. Such as

```
a := 1.0;
```

The 1.0 is a constant as far as TP is concerned. When TP generates code for normal reals, it appears that the constant value is stored in the code itself. When double or extended reals are used, TP does not put the constant into the code itself. The constant value is stored at the top of the procedure that the constant appears in. The code

```
procedure test;
var a : extended;
begin
  a := 1.0;
end;
```

TP generates the code.

```
0000803F       -        1.0
55             -        push bp
89E5           -        mov bp,sp
B80A00         -        mov ax, 000A
9A7C02AF5C          -        call 5CAF:027C
83EC0A         -        sub sp,000A
CD3C99060000   -        fld cs:dword ptr[0000]
```

The mnemonic FLD loads the real constant into the 80x87 or emulator.

The constant is located at cs:dword ptr[0000] which when disassembled, is the first line of this code segment. The main code has a call statement CALL 0004 which calls this
procedure code thus bypassing the real value.

Now what is means to us is, we should always include the compiler directive {$N+,E+,F+} in our program. It is not always needed obviously, but it is safe to include it. If you write an application that uses reals, there may be an error.

The system code has been tested on machines with and without numerical coprocessors. One test included some workers with coprocessors and some without. The above compiler directive worked for both setups.

Units and System Code

There are a considerable number of routines that are needed to perform the operations of the Parallel Lan System. By
incorporating these routines into a single Turbo Pascal Unit, we are able to successfully hide them from the developer.

Turbo Pascal requires units to be included in a program by using the command USES. Programs intended to execute on the PLS should start with the code:

```
Program ..........

{$N+,E+,F+}

Uses work, both;
```

There are two units for the PLS; work and both and should be in the directory where the program being written is. By USEing this unit, we have access to procedures and variables that we will use directly in the writing of our parallel programs. Now in the working directory, there are two different version of the work and both units. Work5 and both5 should be used when using Turbo Pascal version 5.x and work6 and both6 should be used when using Turbo Pascal 6.0.

Startup Procedure

We have developed a parallel environment which operates in coordination with Turbo Pascal. Because of this, there are several problems that must be dealt with directly. To solve several problems, a parallel program must include, what we call, a startup procedure. The purpose of this procedure is two-fold. First is to initialize the system. The second procedure of the startup procedure is to border procedure which are destined to be sent to worker processes.

Turbo Pascal creates, as far as I can tell, procedures by first coding all real constants and placing them into the code. So if a statement in the language like

```
a := 1.0;
```

Turbo Pascal will encode the 1.0 into the code such as

```
0000 3F55 ( or something like this )
```

After the constants, code is generated to set up the stack for any formal or local variables.

```
            push bp
            mov  bp,sp
            mov  ax, ----  ( the number of bytes required for variables )
call ----
```

There one or two calls to Turbo Pascal procedures. It is my guess that the stack is checked for overflow as well as other housekeeping activities associated with the stack. The code to perform the function so the procedure are generated next. After this code, the stack is returned to its original state.   The last code generated is a return instruction. The 80x86 family of microprocessors have several different return opcodes. These return opcodes are for near, far, and interrupt routines.   Our system requires all procedure and function calls to be generated as far. The reason for this is so Turbo Pascal will always generate a RETF instruction which us $CB hex. By always generating this particular instruction, the system software can make a fairly good guess at where a procedure ends and another begins. This, if we wanted to generate code for the procedures.

```
procedure startup

begin
end;

procedure tosend;

begin
end;
```

The code generated would look something like

```
( procedure startup )

$55  push bp
     mov  sp,bp
     mov  ax,----
     call

     .
     .
     .

$CB  retf

( procedure tosend )
$55  push bp
     mov  sp,bp
     mov  ax,----
     call

     .
     .
     .

$CB  retf
```

If our code was going to send the procedure tosend to a worker node, we would grab the starting location of the procedure, say 0030 hex, by using the @ operator of Turbo Pascal. The system code backtracks until it finds a $CB value. What we are doing is looking for any real constants. The memory from the start of a procedure back to the next procedure's end $CB is copied to the end of the code we are sending to the workers.

Now why the startup procedure. If we did not have a procedure coded before the procedure we are sending to the nodes, the system could search for a long time before another $CB is encountered. By incorporating the startup procedure, we are guarantee to find a $CB. The turbo debugger invaluable when trying to determine exactly what Turbo Pascal does during code generation.

The startup procedure should be included in the developer code right after the compiler directives.

```
Program ...............
```

```
{$N+,E+,F+}

Procedure startup;

begin

end;
```

Now that we have the basics of the startup procedure, we need to put some code into it. The first two lines of the procedure are

```
exitsave := exitproc;
exitproc := @myexit;
```

These lines of code link our exit procedure into the system exit procedure which is part of any Turbo Pascal program. In the event of a run-time error, our exit routine will do some internal memory deallocation and other system functions that allow the system to fail gracefully.

The next size lines of code identify which machine is the master. Ethernet addresses are codes as six bytes which as 00 00 C0 05 86 24 hex. Find out the address of the ethernet card the master system software will execute on using the pktinfo program included with the system. The lines of code necessary to identify the master to the developer program looks like

```
master[x] := $yy;
```

There should be six lines identical to the above with x ranging from 1 to 6. The $ operator identifies the following two
characters are hexidecimal.

The last line of code in the startup procedure is

```
init_system;
```

The system initialization code has been put into this single call in order to keep the startup procedure as simple as possible.

Procedure Declarations

The power of the Parallel Lan System lies in the ability to send procedures to worker nodes. The system is not limited to just sending of a single procedure. Any number of procedures can be sent to workers. There are several rules that have to be followed when designing procedure which will be sent to workers.

Global Variables

When Turbo Pascal compiles a program, space is set aside in a separate data segment in the PC's memory for variables. The memory location of a specific variable is recorded in the compiled code. If a global variable is reference in a procedure sent to a worker, the value obtained when this variable is used will not be the value that we want. The contents of the memory location referenced will be undetermined because the worker was not aware that of that particular global variable.

Worse yet is the assignment of a global variable. The value assigned to the variable will be placed in the workers memory somewhere. It is possible that the assignment will cause the worker to fail.

Global variables should be passed by the developer to worker machine through the Linda instructions and the tuple space.

Readln/Writeln

Procedures sent to workers cannot have readln or writeln statements in them. The reason for this is Turbo Pascal treats the screen and keyboard as files. If used, a FILE NOT OPENED error will appear and the worker will fail.

Formal Parameters

There can not be formal parameters in the procedures sent to workers. Because we are using a version of the Linda coordination language, there is no need for formal parameters. The function of formal parameters are performed by Linda instructions.

Nesting Procedures

Turbo Pascal does not follow the source code exactly when generating code. One such case is nesting procedures. Logic would dictate that the code generated would model that of the source, where by the procedure code generated by the compiler would include procedure nesting. Turbo Pascal does not do this, nor should it. Nested procedures are coded as single procedures and a call in made to them as needed.   This however causes us some problems. When our system generates tuples to send to workers, it is performed at runtime, therefore we do not have access to the symbol table. We cannot determine which procedure to send and which not to. The end result is we are unable to handle nested procedures with this release of the system.

Multiple Procedures

There are programs which are prone to decomposition requiring multiple procedures. The system is setup such that procedure sent to workers can coexist with normal procedures. The developer can use normal procedures during the execution of a program. When a program has more than just a single procedure to be sent to the worker nodes, order is not important. The system will locate the appropriate procedure when asked for.

Say we have a program which requires two different procedure to be sent to workers. Can the developer program or programmer determine which worker will get which procedure? No, we can not determine in what order the workers will get the procedures. If it is important, a system of semaphores could be used.

CAUTION!

There is one important note that must be made about the code that can appear in a worker procedure. Turbo Pascal performs smart linking when ever a program is compiled. In other words, if you use the sqrt library function, it is extracted from the system unit and placed in your code. The entire system unit is not compiled with your code. Therefore, STATEMENTS FROM THE SYSTEM UNIT CANNOT BE USED IN YOUR WORKER PROCEDURES! The reason for this is the worker software cannot effectively be compiled with all of the functions included in the system unit. Once again, this is a limitation brought about from building this system on top of the conventions of a pre-existing compiler.

Developer Program Design

Parallel programming is no easy nor is it straightforward. In this section, I would like to give a few points on understanding the task at hand. This is by no means a tutorial on parallel programming.

All parallel system are different. Then again so are all programming languages. Bit as computer scientists, we are required to be able to learn new programming languages easily. Learning to program the Parallel Lan System can be an uncomplicated task if the problem we want to solve is correctly decomposed.

The Parallel Lan System is designed as a distributed structure machine. The information or data structures we develop for our problem are kept on a single machine, the master. In order to access this information, a request is sent to the master by way of a tuple. There can be potentially a large amount of information passed among the different systems. The designer of a parallel program on the PLS must keep this i mind. Take for instance on the of the example programs described in the main documentation. Mandelbrot is typically designed as a sequential program which executes on a single pixel at a time. When writing Mandelbrot for a parallel machine, several options are available. Do we want each processor to execute the algorithm on a single pixel before returning result, a column, a row , or multiple of each. The main documentation describe the result . But suffice it to say, executing on a single pixel at a time will achieve parallelism because some number of processors will be calculating a pixel value at any given moment. Bit is not optimal. There is too much communication between processors. We can achieve better results with a particular number of processors by increasing to a column or more.

This brings up another point, there are times when adding processors to the system will result in a degradation of the system as a whole.

Tuning

There will almost always be some parameters in a parallel program that will judge the ultimate speed of the program on the system. It must be kept in mind that the key to parallel
programming is to get an initial program up and running. After the program is running, test it and compare to other platforms.

A sequence program executing on a single processor

A one - worker Parallel Lan System

A two - worker Parallel Lan System

A four - worker Parallel Lan System

An eight - worker Parallel Lan System

Graph the results (using execution time) on a chart. This will help illustrate the speed up achieved from going to parallel execution. Do not be surprised if several of the tests are slower than a sequential program. At some point, adding more processor should achieve results faster than the sequential program. A rule of thumb can be used in general. If after executing the algorithm on a four processor system the results are not achieved faster than a sequential program, something is wrong with the design of the program. But before submitting to a redesign, check the establish or try to establish tunable parameters. There are always some variables or aspects of the algorithm used in the program which can be tuned to achieve faster results.

The pixels was a tunable parameter in the Mandelbrot program. In a fiance program it could be the number of variables used in a regression or the number of data points examined. In the linear algebra program shown in the main documentation, its the number of columns calculated by each processor.

Debugging

A parallel program can sometimes be more of a challenge than writing the parallel program itself.   As of this release there are no debugging facilities, but the source code is provided so you can add any support you feel necessary. My only remark on this subject is to run your program on a one worker system and count out pieces of the code until you have identified where the problem occurs. Then run on a two worker system and do the same thing. Just remember the you have many different tasks executing at the same time. A key to debugging a parallel program is to understand the relationship between these different tasks.

Main

In the main section of the developer program, the first statement should be a call to start_up.   Your code for the actual program goes next. After the code, the statement close_system ends the main section.

```
begin

  start_up;

{ user code }

  close_system;

end.
```

Skeleton Code

The following is a skeleton of the necessary code for a developer program. It should reproduced and placed in a .PLS file for easy access. The easiest way for keeping the code separate is to name this code skel.pls and perform a copy when developing a new application.

```
program ------;

{$N+,E+,F+}

uses both, work;  { remember: append 5 or 6 to the end of work and both
depending on the version of tp }

procedure start_up;

begin

 exitsave := exitproc;
 exitproc := @myexit;

 master[1] := $--;
 master[2] := $--;
 master[3] := $--;
 master[4] := $--;
 master[5] := $--;
 master[6] := $--;

 init_system;

end;

{ user procedures }

begin

 start_up;

{ user code }

 close_system;

end.
```

What the Developer Doesn't Show

The developer is an independent program. You will not see the same displays as the master and worker have. Therefore, there should be some activity on the developers screen. The programmer will have to provide this activity. The example programs are all graphical in nature therefore it is quite easy to see that everything is going normally. The programmer will want some kind of indication from the developer as to the state of the system as far as the developer is concerned.

RELAX!

The most important thing to remember when developing parallel programs is to relax.   Parallel programming takes time and patience. Parallel programs have to be planned, in my opinion, much more than sequential programs. The results for this is that in a sequential program, one instruction is going to follow another. Even in the case of decisions and loops.   There is no question about it. It can be traced and reproduced many times. (Unless there is a subtle bug in the program that occurs only every three thousand iterations). In a parallel program, there is absolutely no guarantee as to the order that instructions will be executed by the different machine. Even if the workers are executing the same piece of code.

I strongly recommend coding the programs presented later and running them. Get comfortable with the system and then
experiment. You can always reset the machines and start the system again if something really messes up.

PCS-Linda

In the above section on LINDA, we touched on the different instructions and either use in coordinating parallel programs. Now we want to look at PCS-Linda. PCS-Linda is the dialetic of Linda that the Parallel Lan System is programmed with; along with Turbo Pascal. In order to get Linda is execute using a pre-existing compiler, several requirements had to be made as far as using Linda was concerned. This section will discuss those requirements.

Linda Conversion Program

Turbo Pascal will not recognize PCS-Linda. Try it once. You will get an error message on any of the six instructions. The Linda Conversion Program (LCP) is a preprocessor for the PLS and PCS-Linda. After you have coded a parallel program using Turbo Pascal and PCSL, you must give the program a name ending with the extension '.PLS'. Once this file is created, you can run the LCP to convert your program. LCP will perform some magic and produce a file with the same name ending with '.PAS'. This file can now be successfully compiled with Turbo Pascal.

The LCP will produce code that Turbo Pascal can recognize for each of the linda commands. If there is an error in the Turbo Pascal code, verify that all variable have been declared and things like this. Always remember to make changes to the '.PLS' file and not the '.PAS' file. For more information about LCP, refer to the document Linda Conversion Program.

Tuple Elements

Tuples in PCS-Linda can consist of zero to six elements. The elements can be one of four different types

* Actuals

* Formals

* Data

* Null

Actuals and formal elements will be discussed next. The last two types, the programmer and user have no control over. When an EVAL instruction is executed, a procedure is sent to the master or worker. This procedure is considered to be a data element. In reality it is an actual but there are special needs that must be met when searching and transferring procedure to and from the different machines. The data type allows for cleaner code. The null type is used in every tuple where there is less than six elements. Elements not used in a tuple are given the type null. The null type uses no memory therefore it is a convenient form of termination for these empty elements.

Actuals - Integer/Real

Using a pre-existing compiler caused several different things to happen in the development of the Parallel Lan System and its associated Linda language. This can be seen when using actuals and formals in the tuples.

Integers

Turbo Pascal has several different types of integers; integer, long, word, byte. Each of these can be used in a tuple but there is a rule to their use. If an integer is to be sent in a tuple, it can be represented in two ways. The first is to put the integer into the tuple

```
('info', 1, 5, 8)
```

The 1, 5 and 8 will be interpreted as integer and stored accordingly. The integer used in the two-complement two-byte integer. Therefore the tuple

```
('info', 1234567)
```

would case a compile error when the final code was compiled. This number is too large for an integer stored in two bytes. The same is true if an single byte was to be sent. It would be stored in two-bytes instead of one. In order to sent the above number, we could use the tuple

```
bigint : long;

bigint := 1234567;

( 'info', &bigint );
```

This tuple would effectively send the large number to the master. We would do the same for a byte

```
smallint : byte;
```

```
        smallint := $32;

        ( 'info', &smallint );
```

Remember that the tuple sent to match either of the above tuples would also have to have the receiving integer declared as either a long or a byte.

REALS

The same situation as above occurs with reals as well. When a real is put into a tuple

```
                    ( 'info', 1.234, 1.4E+3 )
```

The real is evaluated as a 6-byte real number. This is an acceptable real number for most applications but in the case of more complex mathematics, a large real may be necessary. Turbo Pascal support reals as either real, single, double, extended, and comp. Each of these tpus can be used just as we did for integers. To send the real number 1.234e+1023 we would use the tuple

```
        bigreal : extended;

        bigreal := 1.234e+1023;

        ( 'info', &bigreal );
```

Again, the variable to receive this real number would have to declared as an extended as well.

Formals

Formals must match the types they are to receive. Because we are programming both the code the worker will receive and the code the developer will execute, we can easily match the types. However, it may happen that in a case where we were using reals and the accuracy was not enough so we change to extended reals, we missed a declaration. The system will not match the tuple because a standard real is 6 bytes and an extended is 10 bytes in length.

INP & RDP

The pre-existing compiler strikes again. The INP and RDP instruction treated as function normally. They will returned either true or false if a tuple was matched by the master. However, we were unable to get the functionality of these two instructions exactly. In PCS-Linda the RDp and INP instructions must be assigned to a boolean variable such as

```
          ok := RDP ( 'info', bigreal );
```

If a tuple was matched, bigreal will contain the real value sent with the tuple and ok will be set to true. If a null tuple was sent by the master, ok will be set to false, and bigreal will not be changed. If we were to use the RDP (INP can be
substituted wherever RDP is used) to control a loop we would do the following

```
ok := rdp ( 'info', bigreal );
counter := 1;
while not ok do
  begin
    inc ( counter );
    ok := rdp ( 'info', bigreal );
  end;
```

This loop would count the number of times the RDP instruction was used before a successful tuple match was found. In Original Linda from Scientific Computing Associates, Inc. this loop would appear as

```
counter := 1;
while not rdp ( 'info', bigreal ) do;
  inc ( counter );
```

This is not a big change but it is one that should be kept in mind when using the RDP and INP instructions.

EVAL

The EVAL instructions is used to send a procedure to be executed by a worker. PCS-Linda only allows one element in the tuple used for the EVAL instruction

* The first element must be &procedure name

The name of the tuple must be 'work'. The worker system software has been precompiled to look for a tuple of this name with the above elements. Because duplicate elements are allowed in tuple space, the name 'work' will not cause a problem when sending more than a single procedure to be executed by workers. Thus the tuples

```
                    eval ( 'work', &compute );

                    eval ( 'work', &result );
```

will not conflict with each other.

Tuple Size

The tuple size of the present Parallel Lan System is 16000 bytes. This will allow a large amount of data to be sent between machine per instruction. If this is not acceptable for your situation, you receive garbage when trying to execute your parallel program, (the workers will probably get confused as your code executes), call us and we can increase your systems tuple size.

What Does OUT (); Produce

Lastly, we would like to show what the instruction

```
        var

          start_col : integer;
          results   : array[1..200] of integer;

        out ( 'col', &start_col, &results );
```

produces when put through the Linda Conversion Program.

```
begin
make_tuple ( 3, 'c', 'o', 'l', ' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ', @start_col, sizeof
(start_col), yes,
@results, sizeof (results), yes,
nil,0,null,
nil,0,null,
nil,0,null,
nil,0,null
);
send_tuple ( out );
end;
```

All of the above is required for a single PCS-Linda instruction. It is beyond the scope of this guide to detail what is being done here. For more information about the technical nature of the Parallel Lan System consult the paper - Parallel Lan System - A Course In Overcoming.

An Example

Let's start using the things we have learned by formulating an example exercise. We want to have multiple processors find the sum of all the integers between 1 and 20. We want to write the program in PCS-Linda and Turbo Pascal. Because of the nature of the PLS, if your system only has a single worker, the code will still execute correctly.

We want our developer to put the numbers 1 through 20 into tuple space and wait for a sum to be computed. the first thing we should consider is what our workers are going to do.

Our workers are suppose to take a number and add it to a sum. The workers are going to get this number and the sum from tuple space. So let's get these two things

```
        in ( 'num', a );
        in ( 'sum', sum );
```

Both a and sum will be declared as integers for this problem. Once the worker has these two things, it will add the number in a to the running sum.

```
        sum := sum + a;
```

After which, the worker has to put the new sum back in tuple space.

```
      out ( 'sum', &sum );
```

The worker is finished. Several questions should come to mind. First of all, why did we IN both a and sum, why not use RD instead.

The first thing that we want to happen is a worker to get the sum, which should be zero since we are starting the addition sequence. The worker should then get one of the numbers to add to the current sum. The first worker will request the current 'sum' tuple and a 'num' tuple. The worker will add the number to the sum and get a value of 1 for sum. This new sum is placed back in tuple space for the next worker. If we did not remove the 'sum' and 'num' tuple, we would have a tuple space that was filled with multiple 'sum' tuples and duplicate 'num' tuples. When the next worker requests a 'sum' and a 'num' tuple, it may get the 'sum' tuple that has a value of 0 and not the one with the value of 1.

The duty of the developer program was to put the work procedure and necessary values in tuple space. The code will look like

```
                out ( 'sum', 0 );
                for i := 1 to 20 do
                  begin
                    eval ( 'work', &worker );
                    out ( 'num', &i );
                  end;
```

The developer will end by INing the ending sum and printing the result.

```
                in ( 'sum', &sum );
```

The program looks like this:

```
        program add;
        {$N+,E+,F+}
        uses work, both;

        var  i, sum : integer;

        procedure startup;
        begin
          exitsave := exitproc;
          exitproc := @myexit;

          master[1] := $--;
          master[2] := $--;
          master[3] := $--;
          master[4] := $--;
          master[5] := $--;
          master[6] := $--;

          init_system;
        end;

        procedure worker;
        var a, sum : integer;
        begin
          in ( 'num' , a );
          in ( 'sum', sum );

          sum := sum + num;

          out ( 'sum', &sum );
        end;
```

```
    begin
      startup;

      out ( 'sum', 0 );

      for i := 1 to 20 do
        begin
          eval ( 'work', &worker );
          out ( 'num', &i );
        end;

      in ( 'sum', sum );
      writeln ( sum );

      close_system;
    end;
```

Analysis

Now look at this code carefully. As soon as the worker receive their procedures to execute, they try to IN two tuples named 'num' and 'sum'. But what does the developer do, it immediately tries to IN the 'sum' tuple. Now we cannot predict who will get the 'sum' tuple first but the developer is in line to get it. We want the developer to IN the 'sum' tuple after all tuples have added the values to sum not while they are currently adding the values.

We must put something into the code which will delay the developer from getting the 'sum' tuple until after all workers have added all values to sum. We could put a delay, time wise, into the developer code. Delays are dangerous though. Too many parameters go into the circumstances surrounding delays. We need a tuple that can be increment when a value is added to sum; like a loop control variable. We can do this fairly easily in PCS-Linda. Let's define a tuple called 'count'

```
        ( 'count', count );
```

Before any values are added to sum, count should be set to zero and placed into the tuple space.

```
        out ( 'count', 0 );
```

The worker code must include instructions to read the 'count' tuple and increment it when a value is added to sum.

```
        in ('count', count );
```

```
                inc ( count );
                out ( 'count, &count );
```

The most important part is up to the developer. We do not want to read the final 'sum' tuple until after all values have been added to sum. We have two instructions that can be used; either IN or RD. If we use RD it will leave the 'count' tuple in the tuple space. So let's use IN. But what do we want to IN. We want to read the final count of 20.

```
                in ( 'count', 20 );
```

This IN instructions will send a tuple to be matched to the master. Nothing will happen in the developer until a match is made with the tuple 'count' having a value of 20.

Final Code

```
        program add;
        {$N+,E+,F+}
        uses work, both;

        var  i, sum, count : integer;

        procedure startup;
        begin
          exitsave := exitproc;
          exitproc := @myexit;

          master[1] := $--;
          master[2] := $--;
          master[3] := $--;
          master[4] := $--;
          master[5] := $--;
          master[6] := $--;

          init_system;
        end;

        procedure worker;
        var a, sum : integer;
        begin
          in ( 'num' , a );
          in ( 'sum', sum );

          sum := sum + num;
```

```
  out ( 'sum', &sum );

  in ( 'count', count );
  inc ( count );
  out ( 'count' , &count );

end;

begin
  startup;

  out ( 'sum', 0 );
  out ( 'count', 0 );

  for i := 1 to 20 do
    begin
      eval ( 'work', &worker );
      out ( 'num', &i );
    end;

  in ( 'count', 20 );

  in ( 'sum', sum );
  writeln ( sum );

  close_system;
end;
```

Mandelbrot

Mathematicians are able to generate wonderful things with numbers. One particular thing created from numbers are pictures. The mandelbrot complex number set is a set of numbers that when put through a particular set of equations generates a picture. This manual is not the place for a detailed description of the mandelbrot calculations. Books on fractals will typically have a detailed look at mandelbrot numbers. We will describe as much as needed for the calculations and the parallel program we will write.

Graphics Screen

In order to plot a picture on a graphics screen, we have to create a relationship between each of the pixels on the screen and a particular complex number. For the mandelbrot program we present, we are going to concentrate on a 320 x 200 portion of the VGA graphics screen available on IBM PCs or compatible.

Since we are going to be using complex number is the calculations, we need to define what a complex number is. A complex number has both a real and an imaginary part for each number. There will always be two number for every complex number. Turbo Pascal does not have a complex number type. Using the TYPE feature of Pascal, we can create a complex number type

```
type
     complex = record
       realp,
       imag   : extended;
     end;
```

The nature of the calculations suggests that we use extended real units in order to achieve the best possible accuracy.

For the mandelbrot calculations, a corner value is selected which acts as a reference point for the calculations

```
bcorner : complex;
```

This corner represented the upper left hand corner. For the standard mandelbrot picture, bcorner is given the value

```
bcorner.realp := -2.0;
```

```
                bcorner.imag  := -1.25;
```

In addition to the corner, we must know the lengths of the sides of the picture both width and height. The value typically used for both the width and height is 2.50. Notice that all of these values can be changed to get different pictures of the mandelbrot set.

The last values needed for the calculations are called gap values. The calculations are such that the precision of the numbers can be changed for different screen resolutions. The calculations use two different gap values, one for the width of the graphics screen and one for the height.

```
                gap1 : extended; {width}
                gap2 : extended; {height}
```

Because all of the calculations will use the same gap values, they can be predefined. The gap values are calculated by taking the length of each side (2.50) and divide it by the number of pixels in each dimension. Therefore we have the values

```
                gap1 := 2.50 / 320;
                gap2 := 2.50 / 200;
```

What these gaps say is there is 2.50/320 space between pixels on the graphics screen being used to display the mandelbrot number set. When higher resolution screens are used, the gap gets smaller and smaller thus enhancing the resolution of the picture.

Sequential Program

We have the basic values for the mandelbrot picture we wish to procedure. The only thing left is to do the actual calculations. The calculation that must be done for each pixel is

```
                z := z2 + bcorner;
```

When the size of z grows to be greater than 4.0, we can stop the calculations. Mandelbrot numbers are characterized by not approaching 4.0. Therefore, these number could be put through the above equation many times. This suggests that we need some sort of variable to control the total number of times we do the
calculation. A good control value would be 50. If a value is put through the calculation 50 times and the size is less than 4.0, the number belongs to the mandelbrot set. Values in the mandelbrot set
are colored black in our picture. Values that reach 4.0 before 50 are colored

different colors.

The actual calculation procedure will appear this way

```
function calculate ( col, row, bcorner, ncomplex ) : integer;

begin
 size := 0.0;
 result := 0;
 original.realp := ncomplex.realp;
 original.imag  := original.imag;

 while ( results < 50 ) and ( size < 4.0 ) do
   begin
     original.realp := sqr(original.realp) - sqr ( original.imag
                  ) + ncomplex.realp
     original.imag  := 2*original.realp*original.imag +
          ncomplex.imag;
     size := sqr ( original.realp ) + sqr ( original.imag );            inc ( result );
   end;
 calculate := result;

end;
```

This function is performed for each pixel in the screen. In a 320 x 200 graphic screen there are 64000 calls to this function. In a large graphics screen such as 1024 x 768, there would be 786,432 calls to the function.

One parameter in the equation not yet examined is ncomplex. Ncomplex is a complex number which represents the actual pixel on the graphics screen. Ncomplex is calculated by adding the distance the pixel is from the corner of the screen

```
         ncomplex.realp := current_column_pixel * gap1 +
                                                bcorner.realp;
                   ncomplex.imag  := current_row_pixel * gap2 +
                            bcorner.imag;
```

So each pixel starts at the bcorner and adds the number of gaps in each direction the pixel is from the upper left corner, thus multiplication by the gaps.

The entire sequential program appears as

```
program mandel;
```

```
{$N+,E+,F+}  (* use 8087 if present or emulate if not *)

uses dos, crt , graph;

type

 complex = record
         realp : extended;
         imag  : extended;
   end;

var

 bcorner,
 ncomplex   : complex;
 ccol,
 crow,
 row,
 column,
 result      : integer;
 gap1,
 gap2,
 side1,
 side2       : extended;

procedure get_coordinates;

begin

 clrscr;
 write ( 'Enter the real part of the lower left corner ( -2.0 ) :');     readln
( bcorner.realp );
 write ( 'Enter the imaginary part of the lower left corner ( -1.25 ) :');     readln
( bcorner.imag );
 write ( 'Enter the length of real edge  ( 2.50 ) :');
   readln ( side1 );
 write ( 'Enter the length of imaginary edge ( 2.50 ) :');     readln ( side2 );
 write ( 'Enter the pixels length of a row ( 320 ) :');
   readln ( row );
 write ( 'Enter the pixels length of a column ( 200 ) :');     readln ( column );

end;

procedure compute_stuff;

begin
```

```
 gap1 := side1 / row;
 gap2 := side2 / column;

end;

procedure prepare_screen;

var

 graphdriver,
 graphmode      : integer;


begin
 graphdriver := vga;
 graphmode   := vgahi;
 initgraph ( graphdriver , graphmode , 'c:\tp' );

end;

procedure plot ( crow , ccol , result : integer );

var
 color : word;

begin

 color := 1;

 if result < 2 then color := 1;
 if result > 2 then color := 9;
 if result > 4 then color := 2;
 if result > 6 then color := 10;
 if result > 8 then color := 4;
 if result > 10 then color := 12;
 if result > 12 then color := 5;
 if result > 14 then color := 13;
 if result > 16 then color := 8;
 if result > 18 then color := 7;
 if result > 20 then color := 0;
 putpixel ( ccol , crow , color );

end;

procedure get_complex;
```

```
begin

 ncomplex.realp := ccol * gap1 + bcorner.realp;
 ncomplex.imag  := crow * gap2 + bcorner.imag;

end;

procedure calculate_mandel;

var

 original : complex;
 size     : extended;

begin

 result := 0;
 original.realp := 0.0;
 original.imag  := 0.0;

 while ( result <= 21 ) and ( size < 4.0 ) do
   begin
     original.realp := original.realp*original.realp -
                  orginal.imag*original.imag + ncomplex.realp;
     original.imag  := 2 * ( original.realp * original.imag )  +
         ncomplex.imag;
         size := original.realp * original.realp + original.imag  *
         original.imag;
     inc ( result );
   end;

end;

begin

 get_coordinates;
 prepare_screen;
 compute_stuff;
 for ccol := 1 to row do
   for crow := 1 to column do
     begin
       get_complex ( crow , ccol , gap1 , ncomplex , bcorner );
calculate_mandel ( ncomplex , result );
       plot ( crow , ccol , result );
     end;
```

end.

This program can be compiled using Turbo Pascal 5.5 or 6.0 and executed to demonstrate the picture that will be produced. The row and column entries can be increased to 640 by 480 if desired.

Parallel Programming Methods

The mandelbrot program represents a large number of tasks which perform the same operations on a large set of data. Each and every pixel in the graphics screen has to be calculation on using the calculate function presented above. There is no way around it. Thus if we have two processor available for work, we can be doing two pixels at an given moment instead of just one. Just think, if we had 64000 processors, all pixels would be calculated at the same time. Think about the speed up.

Our job is to write a parallel program using PCS-Linda and Turbo Pascal that will perform the mandelbrot program using any given number of processors.

Developer

The easiest way to approach this problem is to first consider the job or duties of the developer. After we have determined the job of the developer, we can look at the code the worker will execute. Recall from above that the calculations for each pixel all rely on a set of simple calculations which were performed only once. These include

number of pixels in each column
number of pixels in each row
length of column
length of row
gap value for column
gap value for row
corner value
column to compute
number of columns to compute

In addition, the graphics screen of the system must also be initialized.

Columns

We must ask a question about how to distribute the work to the workers. The sequential program computed pixels one at a time. Each pixel is then plotted on the screen. We could allow each worker to only compute one pixel at a time. Because of the nature of the Parallel Lan System, there would be some communication overhead for each pixel. In reality it might take longer for a number of processors to compute the screen than it would for a single processor if we only allow each worker to compute one pixel.

A solution to this problem may be to allow each worker to compute an entire column of pixels. This would significantly cut down on the communication because there would only be one exchange for every 200 pixels if our example problem. This is a much better solution. By allowing each worker to computer a column of pixels, we can introduce another value that will tell each worker how many columns to compute. We might run tests to determine if each worker should have one column, two, four, or more between
communication calls.

Results

The developer is responsible for setting up the calculations as well as receiving the results.   As each worker finishes its column or columns, it will have to put the results into the tuple space for the developer to access. Once the developer has a set of results, it can pass to them to a procedure to be plotted. After 320 result packets have been received, the developer can perform any housecleaning necessary and quit execution.

The code for the developer looks like this

```
procedure plot ( col : integer; results : resu );

var i,j   : integer;
   color : word;

begin



for j := 0 to num_col-1 do
 for i := 1 to 200 do
   begin
     if results[i+j*200] < 20 then color := 1;
     if results[i+j*200] > 20 then color := 9;
     if results[i+j*200] > 40 then color := 2;
     if results[i+j*200] > 60 then color := 10;
     if results[i+j*200] > 80 then color := 4;
     if results[i+j*200] > 100 then color := 12;
     if results[i+j*200] > 120 then color := 5;
```

```
      if results[i+j*200] > 140 then color := 13;
      if results[i+j*200] > 160 then color := 8;
      if results[i+j*200] > 180 then color := 7;
      if results[i+j*200] > 200 then color := 0;

      putpixel ( col+j , i, color );
    end;
end;

begin

 start_up;

 graphdriver := vga;
 graphmode   := vgahi;
 initgraph ( graphdriver, graphmode, 'a:' );

 gap1 :=  2.50 / 320;
 gap2 :=  2.50 / 200;
 bcorner.realp := -2.0;
 bcorner.imag  := -1.25;

 cur_col :=    1;
 tot_col :=  320;
 pix_col :=  200;
 num_col :=    1;

 for i := 1 to num_proc do
   begin
     eval ( 'work', &adder );
     out ( 'stuff', &gap1, &gap2, &bcorner );
   end;

 out ( 'screen', &cur_col, &tot_col, &pix_col, &num_col );
 for i := 1 to tot_col div num_col do
   begin
     in ( 'col', start_col, results );
     plot ( start_col, results );
   end;

 readln;


 { system_shutdown }
 close_system
```

```
end.
```

Analysis

The developer starts by calling the startup procedure for initialization of the network.   After the network is setup, the screen must be changed to graphics mode.

```
                graphdriver := vga;
                graphmode   := vgahi;
                initgraph ( graphdriver, graphmode, 'a:' );
```

These Turbo Pascal commands instruct the graphic card in the PC to switch to VGA mode. The Turbo Pascal BGI file for VGA must be on the drive specified in the initgraph statement.

The developer proceeds to establish the constants for the program. Gap1, gap2, and bcorner are the same as in the
sequential program.

```
                gap1 := 2.50 / 320;
                gap2 := 2.50 / 200;
                bcorner.realp := -2.0;
                bcorner.imag  := -1.25;
```

The next four statements set up the working environment for the workers.

```
                cur_col := 1;
                tot_col := 320;
                pix_col := 200;
                num_col := 1;
```

Cur_col is the number of the next column that needs to be computed. Tot_col is the total number of columns that are to be computed. Pix_col is the total number of pixels in each column. Num_col is the step value or the number of columns each worker is suppose to compute.   These values will all be shared by the different workers in the system.

Once all of the values are computed, the developer is ready to put the work and initialization values in tuple space for the workers to pick up.

```
                for i := 1 to num_proc do
                begin
```

```
                  eval ( 'work', &adder );                                    out  ( 'stuff',
&gap1, &gap2, &bcorner );                     end;
```

Num_proc is the total number of processor on the system or the number of processors to be utilized in the calculations. The common values for the workers is the next tuple put into tuple space.

```
        out ( 'screen', &cur_col, &tot_col, &pix_col, &num_col );
```

The developer is now free to concentrate on its own activities mainly receiving and plotting the results.

```
           for i := 1 to tot_col DIV num_col do
                    begin
              in ( 'col', start_col, results );
              plot ( start_col, results );
            end;
```

Num_col is used to divide the total number of 'col' tuples to receive based on the number of columns each worker will calculate.

The last thing the developer needs to do is clean up the tuple space and shutdown the system

```
         in ( 'screen', cur_col, tot_col, pix_col, num_col );              close_system.
```

Worker

Next we must determine what the duty of the worker is. The worker must IN the common data from tuple space. This data would include the gaps and corner value. The worker would then IN the screen information. It is this information that will determine whether or not a column needs to be computed. If the worker IN the tuple and the cur_col to compute is greater than the tot_col value, then the system has successfully computed all columns.   The worker ca quit executing this procedure. If the cur_col is not greater than the tot_col value, it will have to determine how many columns to compute and compute them. The code for the worker looks like this

```
procedure adder;


type

 complex = record
   realp,
   imag   : double;
 end;

var


 plen : integer;
 pkt  : pointer;
 a_tuple : tuple_pointer;

 gap1,gap2,
 a,b,c,
 size      : double;
 bcorner,
 ncomplex,
 original  : complex;
 cur_col,
 tot_col,
 pix_col,
 num_col,
 cc,
 row,
 r,
 indexc,
 result,
 start_col,
 end_col   : integer;
 results   : array[1..200*num_col] of integer;
 finished  : boolean;

begin


 in ( 'stuff', gap1, gap2, bcorner );

 in ( 'screen', cur_col, tot_col, pix_col, num_col );
 if cur_col > tot_col then
   begin
     finished := true;
   end
 else
   begin
     finished := false;
     start_col := cur_col;
     end_col := cur_col + num_col - 1;
     cur_col := cur_col + num_col;
```

```
    end;

    out ( 'screen', &cur_col, &tot_col, &pix_col, &num_col );

  while not finished do
    begin
      indexc := 0;
      for cc := start_col to end_col do
        begin
          for r := 1 to pix_col do
            begin
              ncomplex.realp := cc * gap1 + bcorner.realp;
              ncomplex.imag  := r * gap2 + bcorner.imag;
              result := 0;
              size := 0.0;
              original.realp := 0.0;
              original.imag  := 0.0;
              while ( result <= 21 ) and ( size < 4.0 ) do
                begin
                  a := original.realp * original.realp;
                  b := original.realp * original.imag;
                  c := original.imag  * original.imag;
                  original.realp := a-c+ncomplex.realp;
                  original.imag  := b+b+ncomplex.imag;
                  size :=
original.realp*original.realp+original.imag*original.imag;                    inc
( result );
                end;
              results[indexc+r] := result;
            end;
          indexc := indexc+pix_col;
        end;
      out ( 'col', &start_col, &results );

      in ( 'screen', cur_col, tot_col, pix_col, num_col );
      if cur_col > tot_col then
        begin
          finished := true;
        end
      else
        begin
          start_col := cur_col;
          end_col := cur_col + num_col-1;
          cur_col := cur_col + num_col;
        end;
```

```
     out ( 'screen', &cur_col, &tot_col, &pix_col, &num_col );     end;

end;
```

The worker begins by INing the common information

```
                        in ( 'stuff', gap1, gap2, bcorner );
```

followed by the screen information

```
                        in ( 'screen', cur_col, tot_col, pix_col, num_col )
```

and determines if there is anything to compute

```
                        if cur_col > tot_col then
```

if there isn't anything to do, the boolean variable finished will be set to true and the compute loop will not be entered. If there is work to do, the worker will obtain its starting columns and put it into the variable start_col. The ending columns will be put into the variable end_col. The cur_col will be increased the number of columns the worker is going to compute. Once all of this has bee determined, the screen tuple is put back into tuple space.

```
         begin
           finished := true;
         end
       else
         begin
           finished := false;
           start_col := cur_col;
           end_col := cur_col + num+col - 1;
           cur_col := cur_col + num+col;
         end;
       out ( 'screen', &cur_col, &tot_col, &pix_col, &num_col );
```

Notice that in either case of the if statement, the screen tuple is put back into the tuple space. If this were not done, the other workers in the system would block because they could not find the screen tuple. Therefore, it must be put back into tuple space.

In order to simplify the way results are sent back to the developer, a single array was used that includes enough space for the total number of pixels a worker computes. This includes the situation where multiple columns are computed. The indexc variable does the job of coordinating where the results will go in the array. The first column of pixels will use the locations 1-200 while the next columns uses the location 201-400, etc. Indexc is incremented by pix_col for every column computed.

Once all columns have been computed for this particular group of columns, they are put into tuple space with the

```
              out ( 'col', &start_col, &results );
```

command. The worker will then IN another screen tuple and begin again.

Complete Code

The complete code for the system is

```
program devman;

{$N+,E+,F+}

uses dos, crt, both, work, graph;

const
 num_colu = 2;
 num_proc = 1;

type
 complex = record
   realp,
   imag   : double;
 end;

 resu = array[1..200*num_colu] of integer;


var
 gap1,
 gap2             : double;
 bcorner           : complex;

 i,
 j,
 cur_col,
 pix_col,
 start_col,
 num_col,
 tot_col          : integer;
 results          : resu;
 graphdriver,
 graphmode        : integer;



procedure start_up;

begin

 exitsave := exitproc;
 exitproc := @myexit;

 master[1] := $00;
 master[2] := $00;
 master[3] := $C0;
 master[4] := $05;
 master[5] := $86;
 master[6] := $24;

 init_system;

end;


procedure adder;


type

 complex = record
   realp,
   imag   : double;
 end;

var
 gap1, gap2,
 a,b,c,
 size      : double;
 bcorner,
 ncomplex,
 original  : complex;
 cur_col,
 tot_col,
 num_col,
 pix_col,
 cc,
 row,
 r,
 indexc,
 result,
 start_col,
 end_col   : integer;
 results   : array[1..200*num_colu] of integer;
 finished  : boolean;

begin


 in ( 'stuff', gap1, gap2, bcorner );

 in ( 'screen', cur_col, tot_col, pix_col, num_col );
 if cur_col > tot_col then
   begin
     finished := true;
   end
 else
   begin
     finished := false;
     start_col := cur_col;
     end_col := cur_col + num_col - 1;
     cur_col := cur_col + num_col;
   end;
   out ( 'screen', &cur_col, &tot_col, &pix_col, &num_col );

 while not finished do
   begin
     indexc := 0;
     for cc := start_col to end_col do
       begin
         for r := 1 to pix_col do
           begin
             ncomplex.realp := cc * gap1 + bcorner.realp;
             ncomplex.imag  := r * gap2 + bcorner.imag;
             result := 0;
             size := 0.0;
             original.realp := 0.0;
             original.imag  := 0.0;
             while ( result <= 21 ) and ( size < 4.0 ) do
               begin
                 a := original.realp * original.realp;
                 b := original.realp * original.imag;
                 c := original.imag  * original.imag;
                 original.realp := a-c+ncomplex.realp;
                 original.imag  := b+b+ncomplex.imag;
                 size :=
original.realp*original.realp+original.imag*original.imag;                   inc
( result );
               end;
             results[indexc+r] := result;
           end;
         indexc := indexc+pix_col;
       end;
     out ( 'col', &start_col, &results );

     in ( 'screen', cur_col, tot_col, pix_col, num_col );
     if cur_col > tot_col then
       begin
         finished := true;
       end
     else
       begin
         start_col := cur_col;
         end_col := cur_col + num_col-1;
         cur_col := cur_col + num_col;
       end;
     out ( 'screen', &cur_col, &tot_col, &pix_col, &num_col );     end;

end;



procedure plot ( col : integer; results : resu );

var i,j   : integer;
   color : word;

begin


for j := 0 to num_col-1 do
 for i := 1 to 200 do
   begin
     if results[i+j*200] < 20 then color := 1;
     if results[i+j*200] > 20 then color := 9;
     if results[i+j*200] > 40 then color := 2;
     if results[i+j*200] > 60 then color := 10;
     if results[i+j*200] > 80 then color := 4;
     if results[i+j*200] > 100 then color := 12;
     if results[i+j*200] > 120 then color := 5;
     if results[i+j*200] > 140 then color := 13;
     if results[i+j*200] > 160 then color := 8;
     if results[i+j*200] > 180 then color := 7;
     if results[i+j*200] > 200 then color := 0;

     putpixel ( col+j , i, color );
   end;

end;




begin

 start_up;

 graphdriver := vga;
 graphmode   := vgahi;
 initgraph ( graphdriver, graphmode, 'a:' );

 gap1 :=  2.50 / 320;
 gap2 :=  2.50 / 200;
 bcorner.realp := -2.0;
 bcorner.imag  := -1.25;

 cur_col :=    1;
 tot_col :=  320;
 pix_col :=  200;
 num_col := num_colu;

 for i := 1 to num_proc do
   begin
     eval ( 'work', &adder );
     out ( 'stuff', &gap1, &gap2, &bcorner );
   end;

 out ( 'screen', &cur_col, &tot_col, &pix_col, &num_col );

 for i := 1 to tot_col div num_col do
   begin
     in ( 'col', start_col, results );
     plot ( start_col, results );
   end;
 in ( 'screen', cur_col, tot_col, pix_col, num_col );

 readln;


 { system_shutdown }
 close_system


end.
```

This code is contained on the samples disk provided with the system. Convert it to standard Turbo Pascal, compile it, and run it for an interesting result. In order to understand what the code is doing exactly, follow the code for a single developer and a single worker. Notice the changes that take place in the screen tuple. This screen tuple is the main controller of the entire program.

Once you have the code operational, try changing some of the initial values to obtain different pictures. The color codes (numbers) used in the plotting routine can also be change to different sequences. These number were taken directly from those in the Turbo Pascal Reference Manual.