MONITORING TOOL FOR PARALLEL PROGRAMS IN TRANSPUTER-BASED MULTICOMPUTERS

Prof J. Duato
and
Prof. M.P. Malumbres
Polytechnic University of Valencia
Faculty of Computer Science
Department of systems engineering,
computers and automatics
Spain

INDEX

1.- INTRODUCTION TO MULTICOMPUTERS.

A multicomputer is a computer that contains several processors,each one with its own local memory, connected through a interconnection network (figure 1.1). This kind of machines can achieve good performance for a wide variety of computational problems.

Figure 1.1

The performances depends on the number of processors or nodes and on the latency of the interconnection network.

Muliprocessors also have several processors. Both, multiprocessors and multicomputers belong to the MIMD class (Multiple Instruction, Multiple Data), according to Flynn' s taxonomy. The main difference between them is the memory organization. Multiprocessors have a shared memory. Then, communication between processors is performed by accessing memory locations. Synchronization requires special hardware semaphores. By contrary, multicomputers perform synchronization and communication by sending messages through the interconnection network.

PHYSICAL ARCHITECTURE.

We can distinguish two fundamental elements in the machine hardware:

*** Node or processor.**

This element performs the computations. Some manufacturers use general purpose microprocessors. Alternatively, processors may include routing circuitry in the same chip. For example, a multicomputer can have nodes of this kind:

T800 (ATW)
M68020 (Ametek 2010)
i80386 (iPSC/2)
i80806 (iPSC 860)

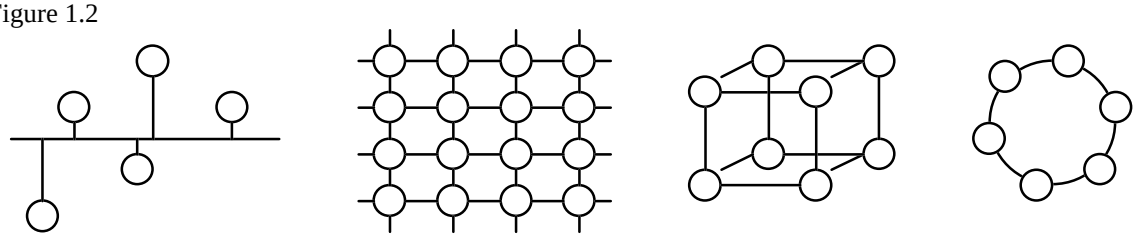
Processors should be powerful enough to make interesting the connection between them. Usually, each node includes floating point support. Sometimes, vector instructions are also supported. However, multicomputer nodes are most distinguished by their network interface. Network supports ranges from software controlled channels to communicate processors.

*** Interconnection network.**

Interconnection networks are a key aspect in the design of high performance multicomputers. Usually, nodes are connected using point-to-point links or channels. As connecting each node with all the other nodes is expensive, each node is directly connected to a small node subset. Then, communication between non-neighbour nodes requires sending messages through intermediate nodes. The main issues in the design of the interconnection network are the following:

- Topology: It determines which nodes are directly connected (figure 1.2). Selecting the optimal topology is a tradeoff between performance and cost. Usually, reducing the distance between implies a higher degree (number of channels per node) and a higher cost. Also, message traffic consumes memory bandwidth, limiting the maximun degree. Recently, physical implementation constraints have also been considered. They include pin count, bisection width, channel bandwidth, wire delay, etc.

Figure 1.2



(a) bus topology *(b) mesh topology* *(c) binary cube* *(d) ring topology*

- Routing algorithm: It determines the path between the source and destination nodes. Generally, this algorithm is distributed, computing the path step-by-step. Static routing algorithms forward messages following the same path for each source-destination pair. Adaptive algorithms can select between alternative paths, usually taking into account network traffic.

- Flow control: It determines how the information advances through the network. The store-and-fordward technique waits until the whole message has been received before routing it. By contrary, the whormhole technique splits messages into small units or flits. As soon as the header flit is received it is routed, reserving the correspondient channels. Flits follow their header in a pipeline fashion. Also, messages do not consume memory bandwidth in intermediate nodes.

In general, the interconnection network is a bottle neck in a multicomputer system. It is not able to serve all messages without a significant delay. By this reason, a well designed network is a goal to achieve good performance and efficiency in this kind of machines.

GRANULARITY OF MULTICOMPUTERS.

The grain of a multicomputer is de fined as the complexity or size of each node. For example, we can say that multicomputers with nodes with several Megabytes of local memory are medium grain machines, while machines with several Kilobytes of node memory are called fine grain multicomputers.

Suppose that we want to design a multicomputer with one Gigabyte of main memory. We can build 256 nodes with 4 Megabytes each one. All the circuits of this node fit in a singel board. This machine would be of medium grain, where each node might be comparable to a workstation. However, we can build a fine grain multicomputer with 32768 nodes with 32 Kilobytes of memory per node. This system can achieve concurrency rates of more than two orders of magnitude than multiprocessors. In this case, the whole node circuitry may fit in a single VLSI chip, increasing the speed of the processor and processor-memory communications.

MULTICOMPUTER EVOLUTION.

The first experimental multicomputer was built during 1981 at Caltech. It was called the "*Cosmic Cube*", a concurrent computer based on message-passing. This system belongs to the first generation of multicomputers. Later, other multicomputers emulated the Cosmic Cube in his architecture and programming system: *iPSC/1, Ametek s/14 and N-Cube 10*. The size of a node in these systems was between seven chips and two boards, and the number of nodes between 4 and 1024.

All of these systems have a software controlled router and use store&fordward flow control. Interconnection networks use the binary n-cube topology also known as *Hypercubes*. This topology has a small diameter, trying to overcome the limitations of store-and-fordware.

The first generation multicomputers are considered of general purpose machines. They have been used in a lot of applications:

- Matrix computations.
- Differential equations.
- Fast Fourier Transforms.
- Heuristic search.
- Circuit simulation.
- Etc.

The current state of the evolution of these machines is the second generation. Nodes have grown in computing power and storing capacity in one order of magnitude (see figure 1.3).

A fundamental difference with their predecessors is the interconnection network. Second generation multicomputers use whormhole flow control, drastically increasing network performance. Whormhole routing decreases message latency (time required to transfer a message), making it almost independent of the distance between nodes. Additionally, routing algorithms are computed by hardware, reducing latency even more. As a consequence, more wirable topologies are prefered. Most second generation multicomputers use a 2D-mesh topology. In this topology, wires are short. Then, channel bandwidth is increased at the expense of increasing the distance between nodes, resulting in faster networks.

Intel and Ametek have developed second generation multicomputers :

Figure 1.3.

- The *iPSC/2* is based on the i80386 microprocessor and a hypercube topology.
- The *Ametek 2010* is based on the M68020 microprocessor and a bidimensional mesh topology.
- The *iPSC-860* is based on the i860 microprocessor, reaching a peak performance of 7.6 Gigaflops
- Intel has also used a bidimensional mesh in the *Touchstone*. This multicomputer has 512 nodes, reaching a peak performance higher than 30 Gigaflops. This computation power is comparable to the most powerful supercomputers.

Because multicomputers are dimensionable in size and technology, the third generation will have minor changes in the structure of the multicomputer and mainly improve the technology of circuit integration. The relation between communications and compute capacity will not change considerably.

The improvements will be localized in the nodes, increasing the compute capacity and making faster the switching of tasks in a multiprogramming environment. Also mew flow control techniques like the use of virtual channels, as well as adaptive routing algorithms are expected to be incorporated in the interconnection network.

2.- PARALLELISM IN THE PROCESS WORLD.

When attention is paid to ordinary processes, it can be observed that these processes have a parallel nature themselves. However, we have always developed sequential programs that try to approach, emulate and imitate these processes.

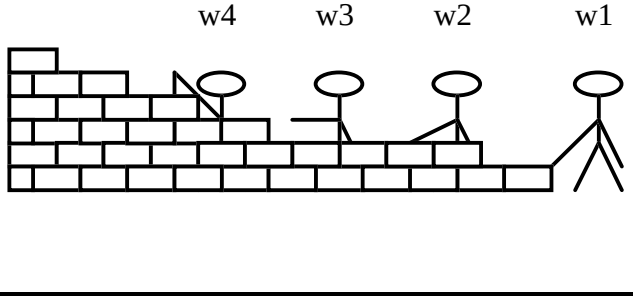
Example of a parallel process:

The problem consists of building a wall of bricks. This example will show the problematic of parallel programming versus sequential programming.

We suppose that we want to build a wall of bricks. If we only have one worker (sequential programming), he will spend one day to complete the wall.

If there are "n" workers assigned to this task, we can build the wall faster. This is obviously parallel programming. The problem with "n" workers is to achieve the synchronization among the workers, without disturbing one another. One approximation for this problem consists of saying to the workers that arrange only one row of bricks (or a group of rows) (figure 2.1).

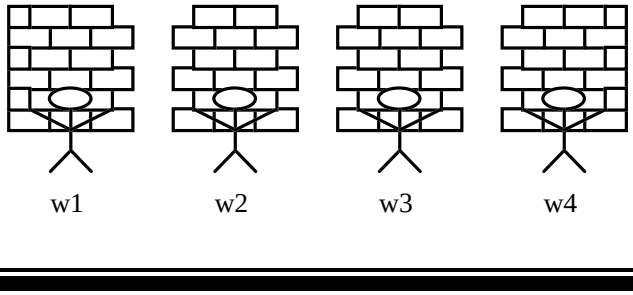
Figure 2.1



The first worker that begins the task must be the worker that arranges the first row. The second worker will arrange the second row, and so on. The second worker ca not begin his work until the first one has arranged the necessary bricks of the first row. This is a pipeline scheme that exploits a certain concurrency. The wall will be terminated in "1/n + k" days.

The other approximation consists of dividing the wall among the workers, so that each worker must build his piece of wall. These pieces of wall are of the same size (figure 2.2).

Figure 2.2



At a first glance, there are not dependencies among the workers, each worker works independently of the others. This is another type of concurrence.

At a second glance, we find some problems:

The interface between pieces of wall require cooperation between the corresponding workers. Not all the pieces of wall are built in the same way. In this case the extremes of the wall are different from the rest.

When we divide a program into a set of processes to execute them in parallel, the results of each process must be joined giving us the final result of the initial program. A certain kind of collaboration is needed between processes to do this. However, there are special cases in some types of processes in which special cases are solved in a different way.

If we make the processes profiting their concurrent nature, we achieve more efficiency and more facility to build them. By this reason, we must considered to programming in parallel.

If we have a process that must be implemented, we can create a sequential program and transform it in a parallel program later. This may be very expensive and tedious and sometimes impossible to get an equivalent version of the sequential program.

3.- PROGRAMMING MULTICOMPUTERS.

Once we have decided to write parallel programs, to fit with our requirements, we need a programming environment in the multicomputer. The operating system must give us a set of primitives that makes programming easier. For example, in the *Ametek 2010* the next functions are defined:

- spawn ("name",n,p,m)
Creates an instance of the compiled program "name" as a node process with the identifier "(n,p)". The process placement is determined by the programmer.

- aspawn ("name",&n,&p,m)
Is the same that spawn with the difference that the process placement is determined by the system.

- ckill (n,p,m)
Kills the node process "(n,p)" and removes any messages that may be queued for it.

- exit()
Kills the calling process and removes any messages that may be queued for it.

- xmalloc(k)
Allocates a message buffer of "k" bytes and returns a pointer to the message buffer.

- xfree(b)
Frees the message buffer pointed to by "b".

- xrecvb()
Returns a pointer to a message buffer, but only after a message arrives for the calling process.

- xrecv()
Is a non blocking version of xrecvb. It may return a NULL pointer if a message has not arrived for the calling process.

-xsend(b,n,p)
Send the contents of the message buffer pointed to by "b" as a message to process "(n,p)" and also frees the message buffer.

- xlength(b)
Returns the length of a message buffer pointed to by "b" in bytes.

- nnodes()
Returns N, the number of system nodes.

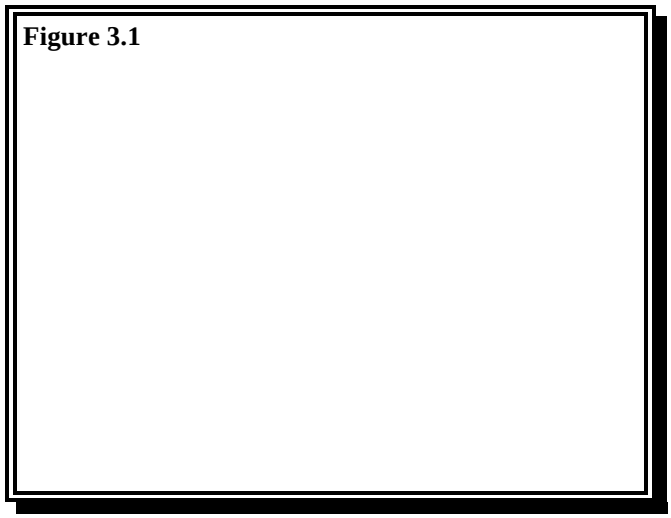
- mynode()
Returns the node part of the identifier of the calling process.

- mypid()
Returns the pid part of the identifier of the calling process.

In general, a process is identified by two parameters: the node where it is running and the pid number. A running process can create or eliminate other processes. With the "spawn" function we decide where we want to execute a process. The fast creation of processes is a very important characteristic for a multicomputer operating system.

Another fundamental issue in a multicomputer is the message-passing system. A message is the basic information unit for process communication. It can have any length, although the message length and destination select different protocols of packing and routing of messages. However, these things are invisible to the programmer.

For sending a message the sender process must have a reference to the receiver process.



In the figure 3.1, a directed graph is represented, where the edges are processes of one application and the arcs represent the references. Messages that travel between two processes can suffer an arbitrary delay due to traffic congestion across the interconnection network .

When a process has to receive a message, we can select a blocking or non blocking reception. This is very interesting and give us flexibility in programming because we can decide if the process must wait for a message (can not do anything) or if it can execute other tasks while the message arrives (non blocking reception).

Now, we are going to present a programming example. The algorithm is a "*mergesort*" that orders a list of elements. This algorithm makes the next:

- A list of length 1 is ordered.
- A list of length 2 or more must be divided in two lists which lengths differ at most in one unit..
- Mergesort is applied recursively over each of these lists. Later, the two ordered lists are mixed in one ordered list.

```
#include <cube/cubedef.h>
typedef struct MESG MESG;
struct MESG {          int pnode, ppid;
                                int tbase;    /* Initialized to 1 */
                                int len;
                                MESG ** type; };
#define BUF(v) ((double *)(v+1))
unsigned int this_node, this_pid, node_cnt;

main()
{          MESG *v;
           this_node = mynode();
           this_pid = mypid();
           node_cnt = nnodes();
           v = (MESG *) xrecvb();
           mergesort(v);
           xsend(v, v->pnode, v->ppid);
}
```

```
mergesort(v)
{       unsigned k1, k2, i, new_node;
        MESG *v1, *v2, *vtemp;
        double *d, *s;
        if (v->len <= 1) return;
        k1 = (v->len + 1)/2;
        k2 = (v->len)/2;
        v1 = (MESG *) xmalloc (sizeof(MESG) + sizeof(double)*k1);
        v2 = (MESG *) xmalloc (sizeof(MESG) + sizeof(double)*k2);
        for (i=v1->len=k1, d=BUF(v1), s=BUF(v);i--;) *d++=*s++;
        for (i=v2->len=k2, d=BUF(v2);i--;) *d++=*s++;
        new_node = this_node ^ v->tbase;
        v1->tbase = v2->tbase = v->tbase << 1;

        if (v1->len >= 20 && new_node < node_cnt)
        {
                spawn("msort", new_node, this_pid, "");
                v1->pnode = this_node;
                v1->ppid = this_pid;
                v1->type = &v1;
                xsend(v1, new_node, this_pid);
                v1 = 0;
        }
        else mergesort(v1);

        mergesort(v2);
        if (!v1)    vtemp = (MESG *) xrecvb();
        else                vtemp = v1;
        Mix_vectors(v, v2, vtemp);
        xfree(v1);
        xfree(v2);
}
```

This algorithm exploits the "divide & conquer" strategy. The recursive calls to the mergesort process would create a binary tree, although it is more elegant and efficient to distribute the load to nodes in a multicomputer. In each call, mergesort divides the list in two equal parts, it sends one of this to other node, and with the other part it calls itself recursively.

In the figure 3.2 we can see how work is distributed among the nodes of a multicomputer in each recursive call. The variable "v->tbase" indicates which is the next node to work.

Figure 3.2

"C" is the root process that makes the first call to mergesort with the entire list (node 0). It divides the list in two parts deciding the destination node of one of them and sending it to this "new_node". The rest of the list is solved in recursive calls to mergesort. At this moment the node 0 and 1 have half the list, they must make the same operation, dividing their lists in two parts and so on. This process terminates when the list size is small enough or there are not free nodes.

Another way of developing parallel programs is to define in the source program the tasks that must perform each node of the network. Once the program has been compiled, we will execute it in all the nodes of the multicomputer. Each node will decide what to do. So, for example:

```
main()
{       int i;

        i = mynode();
        switch (i)
        {
                case 1: do_something();
                case 2: do_nothing();
                case 3: do_computations();
                ....
                case 8: do_all();
        }
        exit(0);
}
```

This kind of parallel program structure is clear and easy to learn. Our programming environment is based on it. This programming style is known as ***SPMD*** (Single Program, Multiple Data) because all the processes execute the same code. "**If**" and "**case**" sentences tailor the code for each particular process. Of course, there exist some algorithms which do not need to tailor the code. Then, all the processes will execute exactly the same code. However, computations may still be tailored by the use of the node pid (*i=mynode( );*). Even when all the processes execute the same code, execution is asynchronous, and execution time may be different for each process (depending on the input data).

4.- THE ATW800 AND THE HELIOS OPERATING SYSTEM.

In previous sections we introduced several aspects about multicomputers, mainly the hardware design and programming. Now we will focus on a particular multicomputer for which we have developed a performance monitor able to capture some information about the execution of parallel programs.

The *ATW800* (Atari Transputer Workstation) is a multicomputer that belongs to the first generation of multicomputers and may be considered as a medium grain multicomputer. It can be accessed through a host that performs I/O operations, controlling several devices such as: keyboard, mouse, disks, printers, etc.

The ATW800 is based on the *T800* transputer from Inmos. In a basic configuration the ATW800 has one T800 (root processor) and can expand up to 12 additional T800. The topology is completely configurable (and can be dynamically reconfigurable), we only have to change the T800 connections to obtain any topology. We have chosen the 3D hypercube topology.

Figure 4.1

So the multicomputer has 9 nodes numbered from 0 to 8, where the node 0 is also called root node (figure 4.1).

The T800 is a 32 bit microprocessor with 4kbytes of internal RAM. It belongs to the RISC family, has a memory interface, a floating point coprocessor and 4 bi-directional serial channels with a bandwidth of 20 Mbps. This processor can achieve 1.5 Mflops. The instruction set includes instructions for creating and ending processes as well as for communication between them.

The processor include microcode for scheduling processes in two priority levels:

- High priority processes: these processes can interrupt other processes of low priority, after any instruction, and execute themselves until terminate their task. This processes are equivalent to interrupt routines in conventional processors.

- Low priority processes: They are scheduled with a *Round-Robin* strategy. The context switch is very fast because always appears over instructions with little information to save (i.e. branch instructions).

The T800 performs the communications among processes using channels. The channels are either memory locations or external links. When two processes want to communicate between them, they must meet in one of these channels and exchange their information.

Each channel is associated to a pair of processes, allowing unidirectional communication. The communication is synchronous. When a process executes an input or output instruction, it is suspended until its partner process executes the corresponding input or output instruction. The processor automatically shedules another process after suspending the communicating process.

The instructions are the same for internal channels and external links. In the first case, when the second process executes the input or output instruction, a memory-to-memory copy is performed. For external links, a direct memory access device is programmed in each processor, transferring the information. As channels are point-to-point, sharing an external link between several processes requires a multiplexor process. This service is usually provided by the operating system.

HELIOS is a distributed operating system designed to work with a parallel processing system. It is *Unix* compatible, showing a user interface similar to Unix but with some extensions for parallel processing.

We discusse the message-passing system and process creation in HELIOS, because we need to know this mechanism to develop the monitoring tool and the programming environment .

Helios implements the message-passing system inside the kernel, using a message port table. Each port is a connection point among processes and contains a communication channel. When processes want to establish a communication, they must meet in a message port. This scheme of communication is called "*Rendez-vous*". When both processes arrive to this message port, the communication begin and a copy of the message is made in the process buffer.

If a process arrives before, it must wait until its partner process arrives. It waits during a time period that is established by the user (timeout). If the timeout is reached then the communication is canceled. If the other process arrives before timeout, then the communication will be established.

Helios supplies a list of low-level primitives for message-passing, mainly : *PutMsg* and *GetMsg*. So, if one process calls the primitives, it suspends its execution until the meeting point has been reached or until timeout.

The message-passing may be local or remote. In other words, the destination process may be in the same node or in another one. In both cases, Helios uses the same primitives.

At last, we must know how a process is executed in a node of the multicomputer. Helios makes this in the following manner: Inside the kernel of each node there is a "Processor Manager" that manages the processor resources. This manager responds to program execution requests that allow processes to be executed in this node under its control.

If you want to execute a process in an arbitrary node, Helios supplies the "Execute" primitive. This primitive requires the name of the executable program and the destination node. When we call the Execute primitive, it finds the executable code and sends a request to remote node´s processor manager. This remote processor manager will receive a copy of this program code and will try to execute it in its processor environment.

5.- A PROGRAMMING ENVIRONMENT FOR MULTICOMPUTERS.

Helios supports three programming strategies:

- Sequential programming: Is the traditional option that has been used in conventional computers. These sequential programs will be executed in one node or processor of our machine.

- Explicit (or outward) parallel programming: There are algorithms that can be split into independent modules. Each module can be executed as a independent task, but all the modules will be interconnected using pipes (like Unix pipes). For example, a compiler environment. This program consists of:

- Preprocessor.
- Initial compilation.
- Final compilation.
- Assembler.
- Linker.

All of these modules can be executed in parallel and they take the input data from pipes and deliver output data to pipes too. If we want to exploit this kind of parallelism, we can execute the modules in different nodes.

- Implicit (or internal) parallel programming: It is the best way to profit the parallelism of a multicomputer. It is not an easy task to develop parallel programs if we are used to programming sequentially. The advantage is obvious, parallel programs will have lower costs than their sequential versions, and the difference will be greater if we use the power of all nodes.

There are a lot of applications and examples of parallel programs that convey this programming philosophy such as: ray tracing, matrix computations, tracking computation systems, chess systems, etc. We follow this kind of parallelism in the next sections.

The programmer task will be easier and faster with a adecuate programming environment, and will not be necessary to know in depth the system hardware and the operating system. If there is not such environment we must create it. This option was taken for the ATW multicomputer.

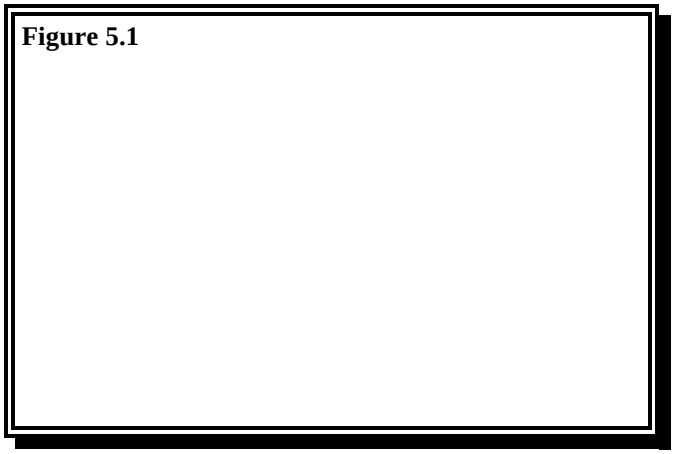
If we intend to build this programming environment, we have to know the message-passing system and the process management used by the operating system (Helios). We want to provide the user with a set of primitives similar to the ones provided by other multicomputers (Ametek 2010, section 3).

The first step is to know how we can execute processes in any node of the network. In the ATW, the "*processor manager*" supervises all processor resources. It is resident in each node of the network and is able to load and execute any process. If there is an executable program, "prog", and we want to execute it in the node number "n", we have to locate the processor manager of node "n" (checking if the node is alive). Then we send it a request by means of the "*Execute*" operating system call.

This mechanism is identical for local and remote processes. Now we can execute programs in any node.

The second step is the message-passing system that supports interprocess communication. Helios provides us two low-level primitives for this purpose: PutMsg and GetMsg. These primitives have one parameter in their call. This parameter is a *MCB* (Message Control Block) that contains the source and destination nodes of the message, a message buffer, the message size, a message port for the rendez-vous, etc.

We are going to design the message format for our applications. This format is illustrated in the figure (5.1).



The header supplies control information and the data consists of an array of bytes, so we can send or receive any data structure; in other words, the data field of the message has an unstruct format.

In the header, there is a field for message type. This field is used to specify several kinds of messages. There are seven types reserved for the monitor, the rest are available for the programmer.

The programming environment must hide from the programmer the management of message ports, data structures, low-level calls ,etc. By this reason, we defined new primitives:

- **Send_msg (proc_id, data_buffer, data_size, timeout);**
- **Rec_msg (proc_id, data_buffer, data_size, timeout);**
- **Broadcast_msg (bit_mask, data_buffer, data_size, timeout);**

where:

proc_id: Reference of the destination node.
data_buffer: A pointer to a memory block where the message data is stored.
data_size: The size (in bytes) of the message.
bit_mask: An 8-bit mask that indicates the nodes which must receive the message.

These primitives return a value indicating if the operation has been successful or not.

Now we need to know how the generation of message ports is made for each process in each node, so that any process could communicate with any other process. This generation is made when the user program is loaded and executed in each multicomputer node. We are going to describe the different stages since the program is compiled until begin the execution in each multicomputer node.

First , we edit and compile our source program. Now, the root node (node 0) loads and executes it in each multicomputer node. When it is executed in one node, it creates a list of local message ports for this one. This list is only known by this node, that must send it to the father (root node) for distribution among other nodes.

When the root node has executed our program in all multicomputer nodes, it begins to distribute the corresponding message ports among them. So, now each node knows how communicate with the others and, by this, we can say that the communications have been established.

Now we need to synchronize all the nodes, so that they begin the execution of the user code at the same time, with a reception of a message from the father (root node) that will arrive when it finishes the distribution of the ports. This message informs the nodes if they must activate the monitoring processes or not (this will be explained in detail in the next section).

At this point, all the nodes begin the execution of the program and the root node is blocked until the nodes complete the program execution.

The environment provides a set of functions:

- **Send_msg (..)**
- **Rec_msg (..)** /* explained previously */
- **Broadcast_msg (..)**
- **int mynode();**
- **int nnodes();**
- **void monitor (int on_off);**
- **Port tx[9], rx[9];**

where:

- **mynode()**: Returns the node that contains to the caller process.
- **nnodes()**: Returns the number of system nodes (9).
- **monitor (on_off)**: This function start/stops the monitoring system when it is called.
- **tx[9]**, **rx[9]**: Lists of message ports for transmission/reception to/from all of multicomputer nodes.

6.- PERFORMANCE MONITOR FOR PARALLEL ALGORITHMS.

This monitor analyzes the execution of programs in multicomputers, giving us information about speed, efficiency, response time, etc. It does not evaluate the performance of the machine hardware. The monitoring processes are included inside of user programs. By this reason, they will record the events generated in each node. This information will be stored after program termination.

This is an off-line monitor, because it shows the results at the end of the user program. It is not difficult to make an interactive monitor that shows captured data while the program is running, but we would introduce an overhead that could give us erroneous monitoring data. The reason being that the monitoring data must be sent to the root node across the interconnection network that is being used by the program.

MONITORING MODES.

We can use the monitor in two modes:

- **Periodic monitorization.**
- **Non periodic or greedy monitorization.**

The *periodic monitorization* consists of a process that periodically awakes and gets information storing it. This process must be very fast to prevent disturbing the execution of the user program. The user must choose the "*sampling*" period for this process.

The *greedy monitorization* records information when it is generated. This monitorization mode uses a limited storing space, because it records all class of activity in each node.

In *greedy mode*, information is captured when a communication event is generated, intercepting the sending or receiving call (PutMsg, GetMsg). In a program that uses the communications intensively, the memory space for storing information may be critical. So, we have made several measures and we have verified that if we saturate the interconnection network with messages, the monitor (in greedy mode) is able to store all the events during three seconds, using 40% of the total memory space of a node.

In *periodic mode*, the information is captured when the monitoring process is activated, averaging the information stored until this moment and during this period. Here there are not storing space problems because when the monitoring process is activated the information is averaged and the associated memory space will be used in the next period.

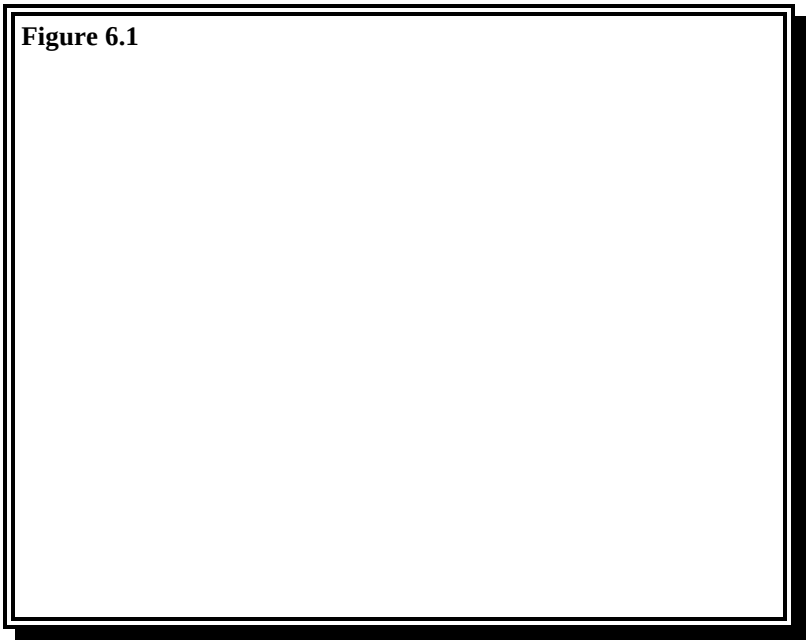
In both modes, the information must be stored in memory, making the capture of information faster. For each event the activation time and the time spent processing it are recorded.

RECORDED INFORMATION.

Two classes of information are recorded:

- **General information.**
- **Detailed information.**

Figure 6.1



General information: Shows us global information about all the nodes. Each node generates one record with the next format (see figure 6.1):

Id -> Identifies the owner of the monitoring data.

Step -> The sampling period (0 if greedy mode).

t_com[9] -> Global communication times of this node with the rest of nodes.

t_cpu -> Cpu time for this node.

t_total -> Total execution time of the user algorithm in this node.

num_msg_send and num_msg_rec -> The number of messages sent/received by this node.

Detailed information: This information contains more especific data. Depending of the monitorization mode used, we have two types of data:

*** Periodic data**: Are generated by a periodic monitorization. Their structure consists of a vector which elements provide information about one sampling period (the first element is associated to the first period, the second element with the second period, and so on). The information recorded in one sampling period is (see figure 6.1):

- **tx_err** -> Error rate sending and receiving messages.
- **num_msg_send and num_msg_rec** -> Number of messages sent and received during this sampling period.
- **processor** -> The owner of this data.
- **t_ovrh** -> Time wasted by the monitor during this sampling period.
- **t_total** -> Sampling period length.
- **t_com[9]** -> Contains the communication times of this node with rest of the nodes during this sampling period.

*** Greedy data**: Are generated by a greedy monitorization and consist of a vector where each element stores information for one event (transmission or reception, see figure 6.1):

- **tx_rx** -> Indicates if it is sending or receiving a message.
- **state** -> Indicates if the message has been sent or received correctly.
- **orig** -> Source node of this message.
- **dest** -> Destination node of this message.
- **com_time** -> Time spent in the communication.
- **stuff** -> Stuffed field.
- **time_init** -> Initial time of this event.
- **data_size** -> Size (in bytes) of the message.

STARTING THE MONITOR.

Suppose that we are in a shell of the operating system that is running in the root node (node 0) and we want to execute our compiled program with the monitor. We must introduce this command:

$ monitor [-on] [-t period] prog_name arg1 arg2 ...

where:

-on -> Activates the monitoring processes.
-t period -> Establishes a sampling period of "period" seconds.
prog_name -> The name of our executable program.
arg1 arg2 ... -> Arguments required by our program.

The monitor command installs the programming environment, executing our program in each node and establishing the communication system (previous section). When the monitor allows all the nodes to begin the execution, it indicates them if there is monitorization and what kind of monitorization must be performed.

Figure 6.2

In the figure 6.2 we show all the steps when starting the monitor from the point of view of the root node, with a directed graph:

State 0: State before starting the execution.

State 1: Initialization of data structures and communications.

State 2: Load and execution of the user program in one node.

State 3: Ports distribution between nodes.

State 4: Waiting state until termination of all the nodes. Monitorization.

State 5: The same as 4 but without monitorization.

State 6: Receives monitorization data from one node.

Figure 6.3

The same steps but from the point of view of a child node (figure 6.3):

State 1: Initialization of data structures and communications.

State 2: Waits for permission from father to retrieve the message port list of all the nodes.

State 3: Receives the message port list and waits for permission from the father to begin the execution with or without monitorization.

State 4: Monitorization and execution of the program.

State 5: The same as 4 but without monitorization.

State 6: When this node has finished, it waits for permission from the father to send it the stored data from monitorization.

State 7: End of execution in this node.

RETRIEVING DATA.

In a parallel algorithm execution, probably not all the nodes will finish at the same time. Then, the root node (father process) must organize the reception of information from the nodes. This reception will not start until all the nodes have finished.

Because the recorded information does not fit in the memory space of the root node, it must be stored in disk. The root node will generate a set of files that will contain all the information. This files are identified as:

- prog_name.mon -> This file contains general information from monitorization.

- prog_name.r00 -> This file contains detailed information from processor 0.

- prog_name.r01 -> Idem but the information is from processor 1.

.....

- prog_name.r08 -> Idem but the information is from processor 8.

Once the monitorization has finished and all the data files have been created, we need something that can represent this information in a graphical form.

7.- USER INTERFACE.

The ATW800 has, like many UNIX machines, a graphical interface based on X-Window that makes the user interface with the monitor and the programming environment more friendly and easy to use. As X-Window is widely spread, we are going to show the different elements of the program interface with several photos.

Figure 7.1

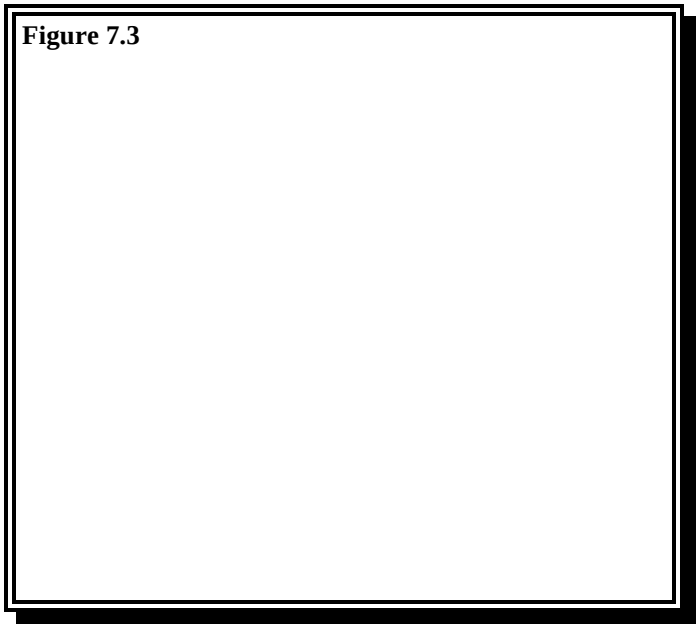
Here is the presentation window that contain a menu bar with several buttons about files, data, and evolution of the monitored program (see figure 7.1).

In the file option, we have a menu pane that describes the different actions that we can perform. For example, we can load the data stored from previous executions, introducing the program name. Also we can run the monitor system, executing a program in the transputer network.

Figure 7.2

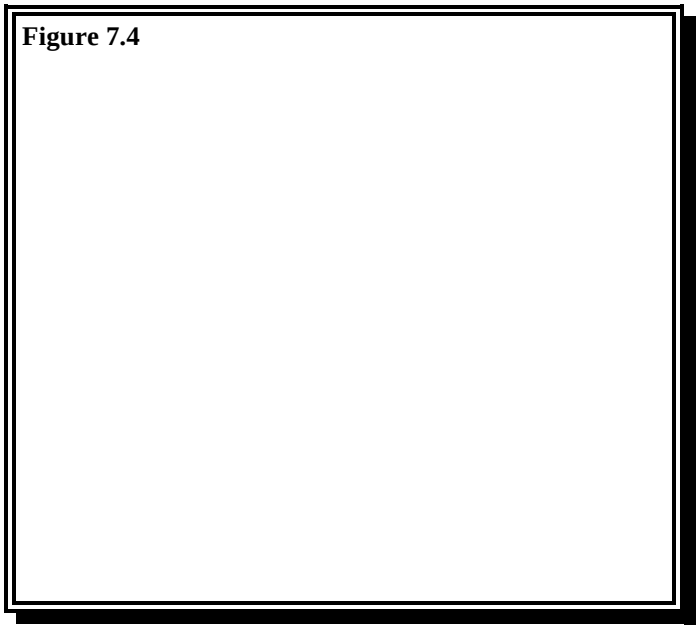
Another menu is the presentation data menu. Here we can represent graphically the monitorization data with bar diagrams and pie charts (figure 7.2 and 7.3). The information data is the same that we have previously described.

Figure 7.3



At last, we have an option that allows us to observe the evolution of the monitorized program, step by step or continuously without pause (figure 7.4).

Figure 7.4



Also we have a set of programs for MS_DOS based computers that can represent graphics with the data files generated by the monitor. These programs analyze the information and create new data files that can be imported from a commercial graphic package (p.e. Harvard Graphics). These programs and the commercial package have been used for drawing the graphics for the next section.

8.- EXAMPLES OF PARALLEL PROGRAMS .

We are going to present two examples that allow us to have a global idea of the items we have presented in this lecture.

Example 1: This program tries to saturate the interconnection network with an intensive generation of messages.

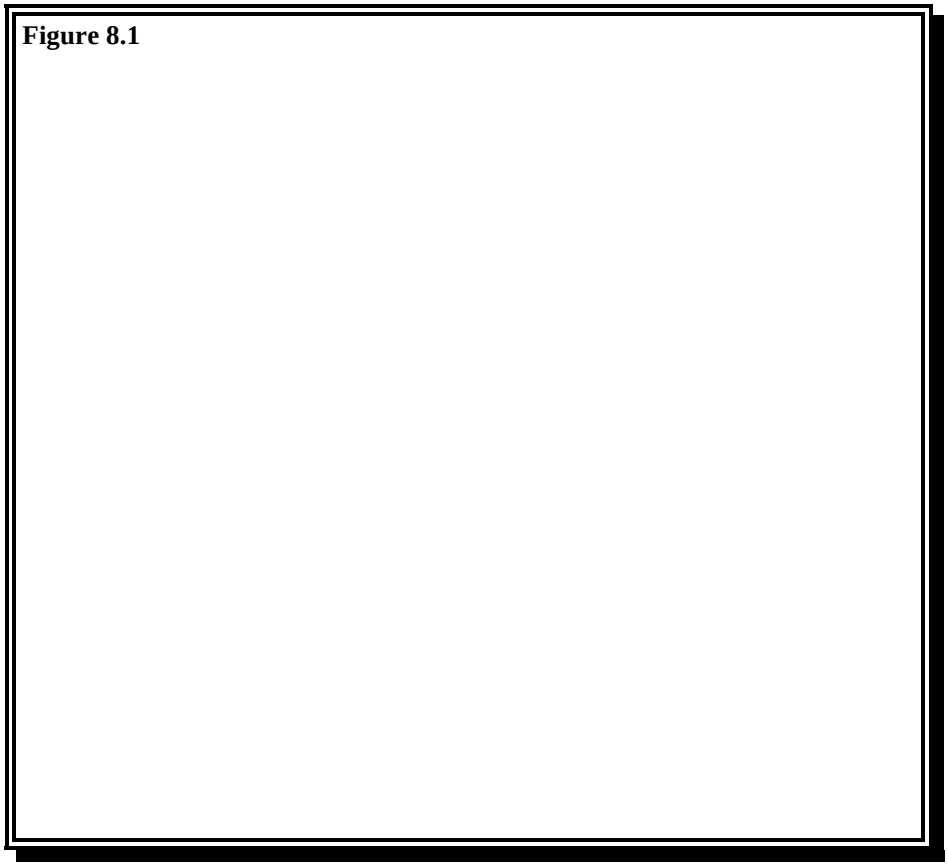
```
#include <string.h>
#include <mon.h>
static char mess[50], mess1[50];

int main(int argc, char **argv)
{               word e;
                int i,j,k,size;

                strcpy(mess, "Hello folks, what are you doing?");
                for (k=1;k<=100;k++)
                 for(i=0;i<9;i++)
                  if (i==cpu)
                  {
                   for(j=0;j<9;j++)
                   if (j!=cpu)
                   {
                                Delay(1000);
                                e=Send_msg(j,(byte*)mess,sizeof(mess),1*SEC);
                                if (e!=0) printf("\n Sending err in node %d",cpu);
                   }
                   else
                   {
                                Delay(1000);
                                e=Rec_msg(i,(byte*)mess1,&tam,1*SEC);
                                if (e!=0) printf("\n Receive err in node %d",cpu);
                   }
                   exit(0);
}
```

Essentially this program works as follows:

for k=1 to 100 do
for i=0 to 8 do
if i = mynode() then send a message to all the nodes
else receive a message from node "i".

Now we compile the program and obtain an executable file ready to be monitored. Then, we execute the monitor command and at its termination we have the data files for this program. In the first figure (figure 8.1) we have represented the general data of this monitorization.

Figure 8.1



In the figure 8.1.c we can observe the communication times between nodes in an 8x8 grid. We can conclude that the farthest nodes from root node have expensive communications in our algorithm due to delays in the interconnection network and to the order in which nodes send messages.

Figure 8.2

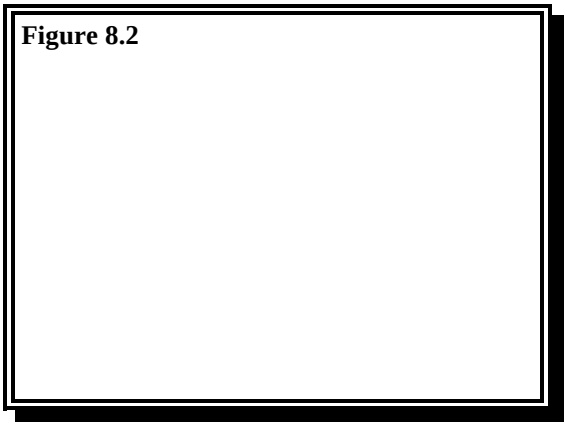
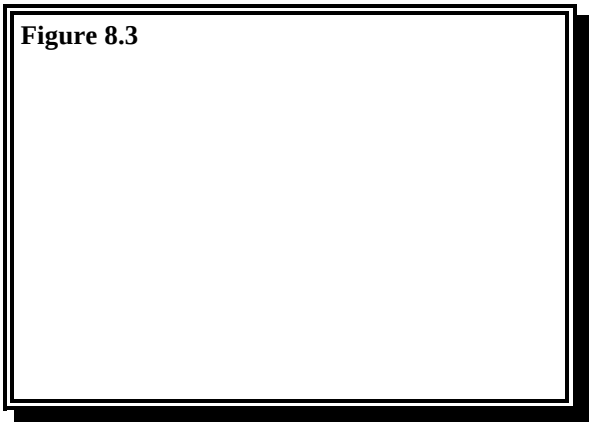


Figure 8.3

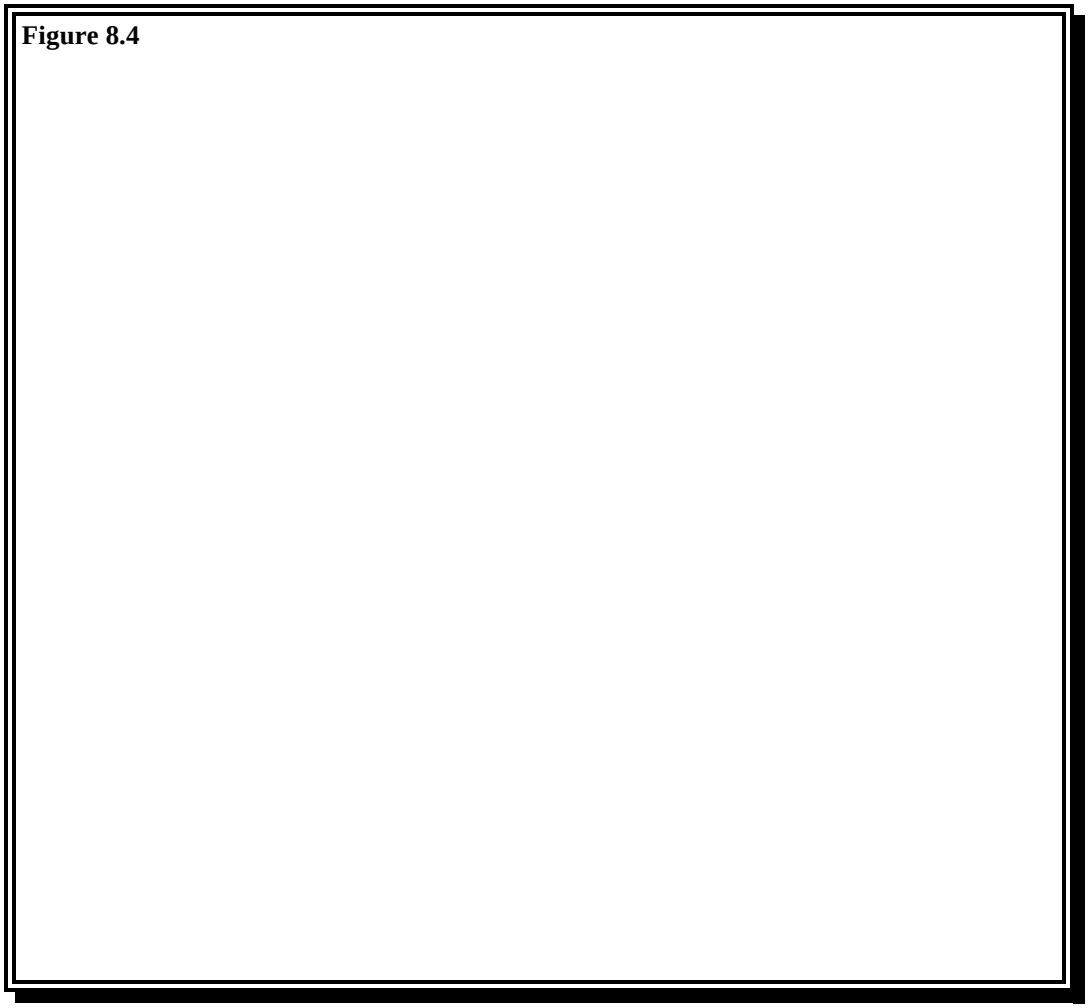


If we perform a greedy monitorization we can show the graphics of figures 8.2 and 8.3:

They tell us the average time for receiving and sending one message from one node to the rest (figure 8.2) and the traffic (in bytes) in one node during all the execution time (figure 8.3).

If we perform a periodic monitorization with a sampling period of 2 seconds, we can show the next graphics in figure 8.4:

Figure 8.4



They indicate us the message traffic in one node and in each sampling period, the overhead introduced by the monitoring process versus communication time in one node, and the communication time of one node with the rest in each sampling period.

Example 2: Resolution of a triangular equation system.

In this example we will need to load the coefficient matrix in all the nodes before the execution. The root node (node 0) will read the matrix from a disk file and will distribute it between the nodes. The distribution strategy is based on allocating, in a cyclic manner, blocks of rows (equations) of the coefficient matrix to the nodes. We assume that blocks contain the same number of rows, and nodes contain the same number of blocks.

Basically, the parallel solution consists of two tasks:

- Resolution of triangular subsystems.
- Updating of independent terms associated to the rest of blocks.

So, in each node we solve an equation subsystem, and the solutions obtained are propagated to nodes that need it. (*The source code of this example can be seen in appendix A*)

If we have a triangular system of 32 rows (equations), and blocks contain 2 equations, the rows will be distributed in the following way:

Node	Block Index	Row Index
1	0-8	0,1 - 16,17
2	1-9	2,3 - 18,19
3	2-10	4,5 - 20,21
4	3-11	6,7 - 22,23
5	4-12	8,9 - 24,25
6	5-13	10,11 - 26,27
7	6-14	12,13 - 28,29
8	7-15	14,15 - 30,31

The program distributes the blocks between nodes with the procedure *DISTRMATRIX*, that reads the matrix from a file. The matrix is stored by rows. When DISTRMATRIX reads a row it computes the node that own this row sending it a message with this row. When DISTRMATRIX ends the distribution, the nodes begin to solve the subsystems.

Figure 8.5

The communications will be made across a logical ring, So one node can only send messages to the previous node in the ring and receive them from the next node (see figure 8.5).

The first node that solves the equation subsystem will be the owner of the last block of the matrix. This node will solve this block and will propagate the solutions to the rest of nodes, which must recompute their independent terms.

One node can be in two states:

(a) It has the current block number. It solves this block of equations (this subsystem), then propagates the solutions and recompute the independent terms.

(b) It does not have the current block number. In this case, it will wait for partial solutions before recomputing the independent terms and propagating the solutions to other nodes.

When there are not more matrix blocks, the algorithm has finished and each node must send to the root node their solutions to form the global solution of the complete equation system .

We have a sequential version of this algorithm and we can compare both algorithms. The *SPEEDUP* is a measure that computes the relation between the execution time of a parallel program and its sequential version. We have run several times both algorithms with different matrix sizes. The results are shown in the figure 8.6.

The next figures show the monitorization results for this algorithm with a matrix of 480x480 elements. Each block has 10 rows.

Figure 8.6

Figure 8.7

General Data:

We can observe in figure 8.7.c that each node only communicates with its successor and predecessor nodes. The reception of messages is more expensive that the emission because of the blocking nature of reception.

Detailed Data:

Greedy monitorization:

Figure 8.8

Figure 8.9

In the figure representing message traffic (figure 8.8) we can see that during 90% of the execution time, the nodes are receiving the elements of the matrix. They are not executing the algorithm, they are only receiving the input data from the root node. This figure clearly shows the importance of a monitoring tool. Without it, people may spend time trying to optimize the algorithm, deciding which is the optimal topology, etc. But forgetting input/output operations. In fact, very few papers about parallel numerical algorithms consider disk accesses.

Periodic monitorization with a sampling period of 2 seconds (figure 8.10).

Figure 8.10

9.- CONCLUSIONS.

With this lecture we have tried to give a general idea about multicomputers, focusing on their programming. Programming multicomputers is not an easy task. However, parallel computers seem to be the only way to overcome the limitations of conventional computers.

In particular, we have studied the ATW800 multicomputer with the target of building a programming environment and a monitoring tool for parallel programs. We can move these tools to any multicomputer. We only need to know its message-passing system and process creation mechanism for beginning to develop parallel programs. Of course, the moving process will be greatly simplified it if the target machine has a graphical interface based on X-Window.

Finally, we have presented two examples of parallel programs, showing the important role played by the monitoring tool. With this tool, it is easy to find where the bottlenecks are.

APENDIX A: Source code for example 2.

BIBLIOGRAPHY

[1] "A survey of parallel computer arquitectures"
Ralph Duncan.
IEEE, Trans. Computers FEB 90.

[2] "Multicomputers: message-passing concurrent computers"
William C. Athas and Charles L. Seitz.
California Institute of Technology, 88.

[3] Computer arquitecture and parallel processing"
Kai Hwang and Fayé A. Briggs.
McGraw-Hill 87.

[4] "Visualizing parallel computer system performance"
Allen D. Malony and Daniel A. Reed.
Department of Computer Science. University of Illinois. SEP 88.

[5] "Monit: A performance monitoring tool for parallel and pseudo-parallel programs"
Teemu Kerola and Herb Schwetman.
Microelectronics and Computer Technology Corporation. ACM 87.

[6] "Configuring parallel programs: The PAR.C, a parallel C compiler"
Dick Pountain.
BYTE, JAN 90.

[7] "IMS T800 arquitecture, technical note"
INMOS Ltdf, Bristol 86

[8] "IMS T800 Transputer, product overview"
INMOS Ltdf, Bristol 86

[9] "Helios operating system"
Perihelion Software, 89
Prentice Hall 89.