Implementation of Virtual Tree Machines

Marius Olafsson
T.A. Marsland

Computing Science Department
University of Alberta
Edmonton T6G 2H1

Technical Report
TR-85-9

ABSTRACT

The report describes an environment for performing experiments in distributed processing. Our system offers researchers an easy way to design, implement and test parallel algorithms. It provides software tools which make possible a variety of tree-structured connections between processes. These process structures are said to form a "Virtual Tree Machine" (implemented on a local area network of VAX 11/780's and SUN-2 processors). We show how these tools have been used both to aid parallel algorithm development and to explore different computer interconnection methods.

[Regenerated from files dated 13 Dec 1987 in /tony/Reports/TR85.9/TR85-9]

17 March 2013

An abridged version of this report entitled "A UNIX BASED VIRTUAL TREE MACHINE", was presented at the CIPS Congress '85, Montreal, June 1985.

Implementation of Virtual Tree Machines

Marius Olafsson
T.A. Marsland

Computing Science Department
University of Alberta
Edmonton T6G 2H1

Technical Report
TR-85-9

1. Introduction

Parallelism may be applied in several ways to increase the processing power available to the execution of a program. These approaches can be broadly categorized into two groups: use of closely coupled or synchronized processors, and loosely coupled or distributed systems. Closely coupled systems have traditionally been more popular since they can be used to speed existing algorithms and even existing programs. For example, powerful vector processors are now well established and most contemporary systems use some degree of pipelining.

One reason for limited progress in experimental computer science is the cost and special purpose nature of the equipment. Specifically, in distributed systems researchers have managed with a collection of connected processors, each with little or no I/O capability, rudimentary operating system support and a small memory. On such systems experiment management is often difficult and the lack of flexibility restricts experiment design. With the widespread use of local area networks, experimenters can take advantage of existing computing facilities, can draw upon the services of a powerful operating system (with such capabilities as virtual memory management) at each node, and can design their distributed algorithms in high-level programming languages. Debugging and monitoring the execution of a distributed program can be improved by using the services provided by the operating system, such as its drivers for various display equipment and its file system. Naturally, all this places certain restrictions on the experiment design and forces careful interpretation of the results, but often these restrictions are not serious and are offset by the advantages.

Another problem with parallel processing in general is the tradeoff between communication speed and the complexity of the connection structure. [1] Tree-structured topologies have been proposed to reduce the connections between processors in distributed systems. [2] The advantage of the tree machine is that the number of links only increases logarithmically with the number of processors, thus making possible the construction of systems with thousands of processors without a prohibitively expensive interconnection network. [3] Another advantage of this architecture is that many problems map naturally into a tree structure. These include NP-complete problems such as various combinatorial methods requiring exhaustive search [4] and tree-searching algorithms. [5] [6]

2. The Virtual Tree Machine

This report describes an environment designed and built to do experiments in distributed processing, using standard equipment and the services of a contemporary operating system to reduce hardware and software costs and to simplify experiment management. We call this environment a Virtual Tree Machine. It is implemented on a network of VAX-11/780's and SUN-2 processors each running the 4.2BSD UNIX® operating system, [7] see Figure 1. In addition, the system includes six standalone Motorola 68000's,

An abridged version of this report entitled "A UNIX BASED VIRTUAL TREE MACHINE", was presented at the CIPS Congress '85, Montreal, June 1985.

without operating system support for use when critical timing measurements must be made. Since these processors can be connected in a tree structured fashion, they are collectively referred to as a Processor Tree Machine (PTM). The only portion of the UNIX kernel that was ported to these six processors was the 4.2BSD networking support. The standalone machines possess no process management and therefore support only one process per physical processor. However, since most of the UNIX run-time library is available, identical code can be run both on the PTM and the UNIX based machines.

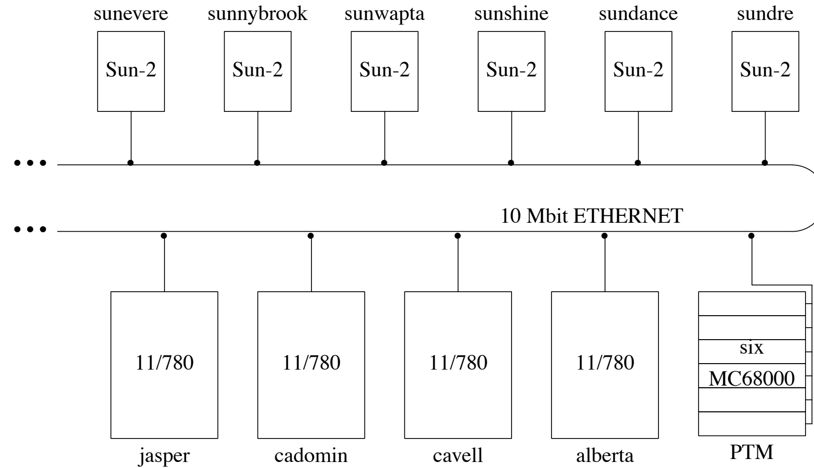


Figure 1. Computing Facilities

The only restriction on the PTM usage as a part of the Virtual Tree Machine environment is that it does not support more than one virtual node per physical processor. The word "virtual" is used here to emphasize that, as opposed to a Tree Machine proper, physical processing elements are replaced by processes under operating system control, while wired connections are replaced by virtual communication paths. The experimenter views the machine as a collection of processing elements - each with ample memory, disks and other I/O devices, and a communication path with its parent and with each of its children. In reality, a VTM is a collection of procedures callable from ordinary user programs and a collection of "node-servers", one on each physical machine. These node-servers receive requests to create nodes of the tree-machine according to the description provided by the user. During development the whole machine might reside on one physical processor before being distributed over the selected physical machines for productive use.

The interface to the VTM is a collection of user procedures, written in C [8] and callable from application programs. These procedures handle connection establishment, connection initialization, exchange of messages, interrupt handling as well as providing information on the configuration and layout of the virtual machine being used. Actually the VTM procedures are based on a more general set of procedures which provide for the creation of arbitrary interconnection topologies of which the tree is only a restrictive subset. These procedures will be described in the next section, but the VTM support routines are as follows:

```
fanout = root_init(name, cfile)
fanout = node_init(name)
```

These are used to give a name to the VTM and establish communications between a parent and its children. `Root_init` is called from the process representing the root of the tree, to read the configuration from a file named in `cfile` and start the recursive creation of the process structure. Each line in the configuration file represents one node in the tree machine. The interconnections between the nodes are inferred from their order of appearance in the file. Thus, the configuration file consists of lines of the following form:

```
host name; #children; bits; file name; input; output; error
```

That is, each line in the configuration file consists of seven fields separated by semicolons(;) as follows: Name of the physical machine on which the node is to run (the host); number of children of the node; trace bits; name of the file containing the node's executable code (full path name from the user's home directory). The last three fields contain the names of files to be opened as the node's standard input, standard output and standard error. Note that if a node has any children, their specification lines must directly follow the parent's specification in the file. For the root node, all except the first three fields may be omitted. The meaning of the trace bits is given in Section 3.1 and an example of a VTM configuration file is in Figure 4-a. `Node_init` creates descendant nodes from the interconnection description received from the parent. Once all communication paths have been created, control returns to the user's application. At each node, `fanout` specifies the number of children created. Later, a parent may send/receive messages to/from its children via the following functions:

```
s_send(child, message, length, interrupt, trace)
s_receive(child, message, length, trace)
```

Similarly a child communicates with its parent with the following:

```
m_send(message, length, interrupt, trace)
m_receive(message, length, trace)
```

Within each node, the children are numbered from 1 to `fanout` as returned from the initialization procedures. The `message` is the address of a buffer containing the bytes to be sent. These four functions return the number of bytes actually sent/received. Messages may be specified to signal their destination on arrival via the `interrupt` parameter. The `trace` argument is used to enable the debugging and message tracing facilities available in the VTM environment, see Section 6.

Facilities also exist for managing the internode signals. These include routines to enable and disable the reception of signals, hold and release signals and to specify the signal handler, as follows:

```
s_int_handler(handler)   s_int_enable()   s_int_disable()   s_int_hold()
s_int_release()
```

where `handler` is the address of an interrupt handling procedure that is invoked whenever a message arrives.

While interrupt driven message management can work quite well, there are always situations when interrupts may be lost (e.g. occur nearly simultaneously) and so extended waits for response may occur. One way to protect against such a problem is for a process to poll another for pending messages. Polling children and parent for outstanding messages is achieved with the following routines:

```
s_poll(children, block)
m_poll(block)
```

where `block` indicates whether or not the call should wait for the next message to arrive. The id's of children with messages in transit are returned in the array `children.` `S_poll` returns a count of children with outstanding messages and `m_poll` returns 1 or 0 depending on whether or not there is a message in transit from the parent.

There are situations where we would like to change the size of the Virtual Tree Machine as the applications proceeds. This could be used for load-balancing, e. g. deleting a node on a heavily loaded machine and recreating it on a machine with lighter load. Also, if a node fails for some reason, a new node can be created in its place thus increasing the fault-tolerance of the whole system. The configuration of the tree may be changed dynamically using the following:

```
add_child(host, name, sin, sout, serr, bits, fanout)
delete_child(child, fanout)
```

where `host` is the name of the physical machine on which the new child is to reside, `name` is the name of the child's executable file. The child's standard input/output streams are renamed `sin, sout` and `serr` If successful, `add_child` will return the new fanout (`fanout + 1`), where `fanout` was the original number of children and `delete_child` will return `fanout - 1`. More detailed information on these routines can be found in Appendix II.

3. Implementation of Virtual Machines

The VTM environment is implemented using a set of basic support routines that allow for the creation of virtual multiprocessors with arbitrary interconnection structures, see Figure 2. These basic routines are in turn built on top of the UNIX networking primitives. The UNIX primitives allow processes to communicate via a variety of protocols and connection strategies. [9] The current implementation of the support routines uses reliable two-way communication channels (called stream-sockets). The semantics of the stream-sockets are similar to UNIX pipes, [7] except that the communicating processes need not reside on the same physical machine. There are two main aspects of the UNIX networking primitives that make them a good basis for implementing a virtual processor system. First, the UNIX inter-process communication model is internally consistent; no distinction is made between interprocess communication and interprocessor communication. That is, the communication processes use the same mechanisms, irrespective of whether they both reside on the same processor or not. Secondly, the client/server model cleanly incorporates the virtual node server so that it needs no special administrative consideration over other servers on a particular machine. Thus, these basic routines provide the tools to implement various virtual environments of which the Virtual Tree Machine is but one. We plan to provide a *Virtual Hyper-Cube* [10] in addition to the Virtual Tree Machine. Users can also elect to use the basic routines directly, to create virtual multi-processors with arbitrary interconnection structures.

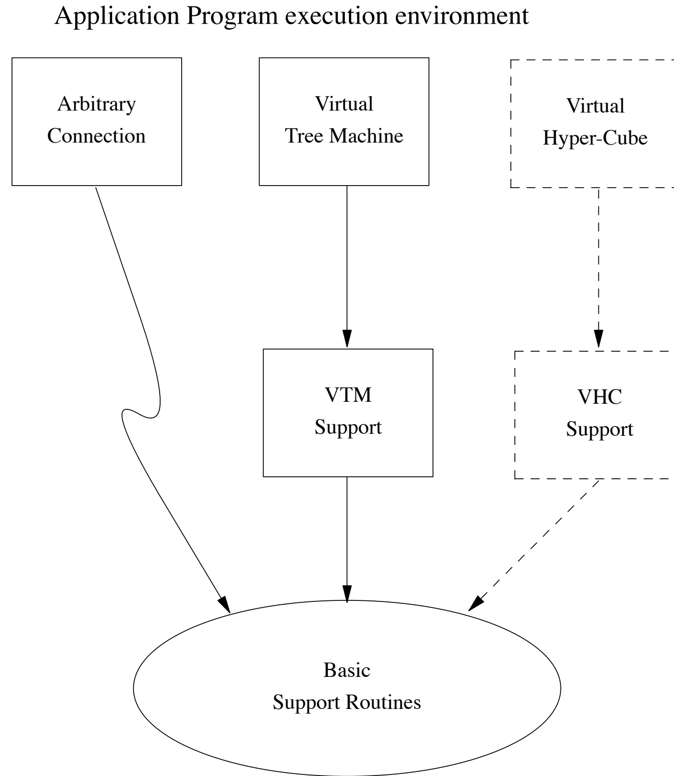


Figure 2. Hierarchy of Virtual Machine Implementation

3.1. Basic Support Routines

In this section we describe the routines that form the bases for the VTM implementation. The execution environment consists of a collection of virtual processors (UNIX processes) whose virtual interconnections (UNIX stream-socket connections over Ethernet) are created on initialization. Each virtual processor (or node) is assigned a unique number *node-id,* which is used to address messages to the node. For obvious reasons, these routines are semantically similar to the VTM primitives discussed in Section 2.

```
NodeInit(Type, MachineName, ConfigFile)
```

This routine initializes the node. There are two types of node initializations, the first (`Type=0`) is used in the node that interacts with the user. The configuration file, `ConfigFile,` is read and other nodes created as specified in the file. The name of the virtual machine is an arbitrary character string pointed to by `MachineName.` All other nodes are of (`Type=1`) and receive the interconnection information from their creator. The configuration file is similar to the one used to define a VTM but is in three parts. The first part is a line containing the number of virtual nodes to be created. The second part is one line per node containing the following information:

```
host name; executable file name; bits; sin; sout; serr;
```

where `host name` is the physical host on which the node is to run; `executable file name` is the full path name of the node (from the user's root directory); `bits` is an integer constant whose bits mean the following:

```
Bit #
0    if set, simple message tracing is on for this node
```

```
1    if set, remote debug is on for this node
2    if set, the node runs on the PTM
```

The three last fields contain the names of files to be opened as the node's standard input, standard output and standard error. The third part of the file is the connection specification. It is a triangular matrix of 0's and 1's where 1 represents a connection. For an example of a basic configuration file layout see Figure 4-b.

The remaining aspects of communication, interrupt handling, polling and reconfiguration are handled in much the same way as for the VTM. For example,

```
SendNode(NodeId, Message, Length, Interrupt, Trace)
RecvNode(NodeId, Message, Length, Trace)
```

have the same semantics as the corresponding s_send and s_receive VTM routines described previously, except that messages may be exchanged with an arbitrary node, `NodeId,` provided a virtual connection already exists. The general interrupt handling routines are as follows:

```
SetHandler(Handler) EnableInt() DisableInt() HoldInt() ReleaseInt()
```

Use of these routines are self-explanatory, but more details are provided in Appendix I. Two routines are provided to check the status of messages from neighboring nodes:

```
PollAll(Nodes, Block) PollNode(NodeId, Block)
```

The node-id's of nodes that have outstanding messages are returned in `Nodes.` Normally, these routines just return 0 if no messages are waiting to be received. Setting `Block` to 1 instructs the routine to wait for the next message to arrive. Finally, two routines are provide to reconfigure the system dynamically at run-time:

```
AddNode(Host, Name, Sin, Sout, Serr, Bits)
AddConnect(Type, NodeId)
```

`AddNode` creates a new process on the designated machine and establishes a communication path, while `AddConnect` creates a connection between two existing nodes. Note that only one connection may exist between two nodes and therefore `AddConnect` cannot be used to provide an additional connection between nodes when one already exists. Further details on the basic support routines are in Appendix I.

4. A VTM Example

As an example, consider the creation of a VTM to execute the configuration of processes depicted in Figure 3.

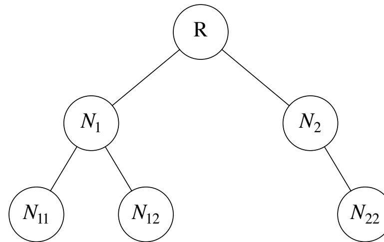


Figure 3. Process Tree

First this configuration must be mapped onto the hardware. There are no restrictions on the number of physical machines that must be available, but for clarity here we map the nodes one per physical machine:

R on sunshine
N_1 on cavell
N_2 on alberta
N_{11} on sunwapta
N_{12} on sunnybrook
N_{22} on sundre

That is, the root resides on a SUN-2 processor, called *sunshine,* the interior nodes are on VAX-11/780 processors and the leaf nodes on SUN's. The mapping between the virtual machine and the physical hardware is described in the configuration file. For the VTM it is a simplified form of the more general configuration file described in the last section. Each line in the tree machine configuration represents a virtual node and contains seven fields separated by semicolons. The first four fields are: the name of the physical machine on which the node is to run (the host); the number of descendants of the node; an integer whose bits provide information to individual nodes (e.g. debug specifications); and the name of the file containing the node's executable code. The other three fields contain the names of files to be opened as the node's standard input, standard output and standard error.

The following configuration file is used to map the virtual processor tree in Figure 3 to the available hardware.

```
sunshine;2;0
cavell;2;0; node -p1;; out1; err1;
sunnybrook;0;0; node -p11;; out11; err11;
sunwapta;0;0; node -p12;; out12; err12;
alberta;1;0; node -p2;; out2; err2;
sundre;0;0; node -p22;; out22; err22;
```

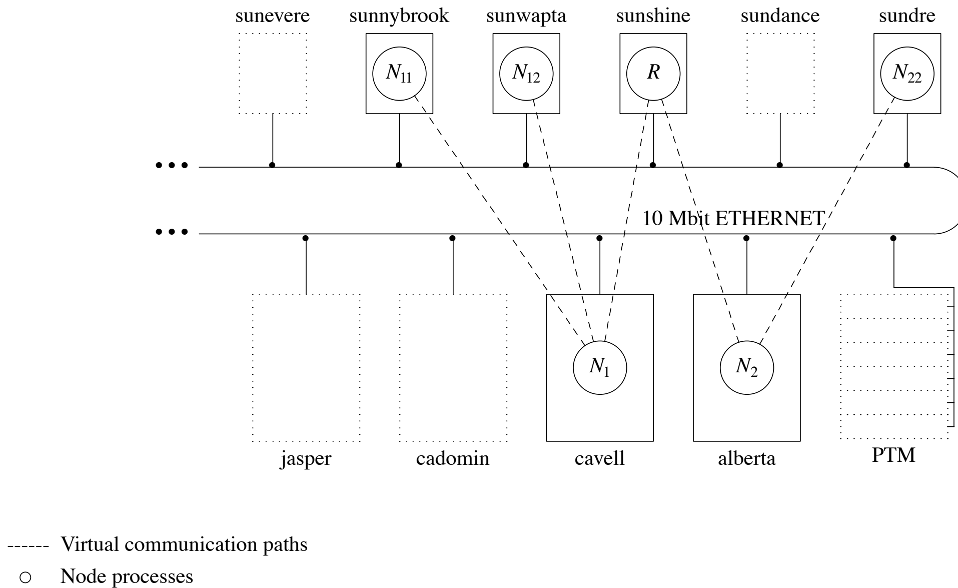
Figure 4-a. VTM Configuration File

When the root process on *sunshine* is started, it calls `root_init` which reads the above form of the configuration file and translates it to the basic configuration file format:

```
6
sunshine  ; 0
cavell    ; 0; node -p1;; out1 ; err1;
sunnybrook; 0; node -p11;; out11; err11;
sunwapta  ; 0; node -p12;; out12; err12;
alberta   ; 0; node -p2;; out2 ; err2;
sundre    ; 0; node -p22;; out22; err22;
0
1 0
0 1 0
0 1 0 0
1 0 0 0 0
0 0 0 0 1 0
```

Figure 4-b. Basic Configuration File

The above configuration is written to a temporary file. `NodeInit,` now takes over to create the specified virtual machine. It sends a service request to the node-server on *cavell*. The server executes the file *node* (from the third field in the configuration entry for the process on *cavell,* prepended with the users home directory), gives it whatever execution parameters specified (in this case "-p1"), and returns to listen for additional service requests. The *node* process on *cavell* receives the configuration from *sunshine* and sees that it has two children. It therefore transmits two requests, one to the server on *sunnybrook* and the other to the server on *sunwapta*. Both nodes see that they have no children and so respond that they were successfully started.



------ Virtual communication paths
○ Node processes

Figure 5. Mapping of Processor Tree onto Hardware Configuration

The interior node on *cavell* then tells the root that all went well. The root now knows that the left branch is complete and transmits a request to the node-server on *alberta* to start up the right branch. Finally, `node_init` returns and the application is ready to start work, since all communication paths have now been established. The VTM created is shown in Figure 5. With this facility several different experiments can be performed at the same time. The Ethernet serves as a shared communication path and processes from different applications could share the same processor.

To illustrate the communication and connection establishment features provided in the VTM environment, the following skeletal C-code segment from an arbitrary interior node is presented below:

```
.
.
fanout = node_init(name);        /* connection to 'fanout' children */
m_receive(buf, n, TRACE);        /* receive from parent */

for(i=1; i<=fanout; i++) {
    s_send(i, buf, length, NOINTS, TRACE); /* send to children      */
}
for(i=1; i<=fanout; i++) {
    s_receive(i, buf, length, TRACE);      /* receive from children */
}
m_send(buf, length, NOINTS, TRACE);        /* reply to parent       */
.
.
```

The process containing the above code segment is invoked by the node-server on its host machine. After invocation, `node_init` waits for the parent to send the configuration of its tree branch. Once received, `node_init` transmits requests to start this node's children (if any). When `node_init` returns, communication has been established with the parent (from which the node receives its work via `m_receive`) and its children (to which it sends some units of work via `s_send`). When this node has finished its work, it receives the results from its descendants (via `s_receive`) and finally transmits its results to its parent (via `m_send`). The parameter *NOINTS* specifies that no interrupts are generated, and *TRACE* is used to specify a string included in a message trace generated by these calls (if any).

This code will be identical on all nodes in the VTM (except the root where communication with the parent would be replaced with user interaction). Thus, every call to `m_receive` has a corresponding `s_send` call in the parent node and every call to `m_send` corresponds to a `s_receive` call in the parent.

5. Communication Speed

The following table shows the effective communication times using the VTM primitives. These results were obtained by measuring the average transit times for messages of varying lengths. The measurements were made for all combinations of physical processors (only four shown below). *PTM/PTM* represents two standalone processors (in the PTM) exchanging messages. *Process/Process* means two processes on the same machine exchanging messages (i.e. the messages do not leave the machine but only go through the software layers down to the networks interface and then *loopback* up through the software layers to the other process). *VAX/VAX* are two VAX machines communicating and *Sun/Sun* two Sun Workstations. In addition, these results are averaged over several days to factor out any peculiar conditions on the network.

As expected, the longer the messages the better effective speed, and the effective byte rate increases linearly with message size up to the underlying packet size (1583 bytes). Notice that these figures include all the software cost, constructing the messages all the way down to packet transmission. Also the cost of decoding a packet of the network, reassembling the messages and delivering the message to the application program. Taking this into account the times represented here are quite respectable.

Table 1. VTM Communication Timing Effective speed in K-Bytes/sec and as % of network bandwidth								
	PTM/PTM		Sun/Sun		VAX/VAX		process/process	
Length	KB/sec	% Raw	KB/sec	% Raw	KB/sec	% Raw	KB/sec	% Raw
4	0.8	0.06	0.5	0.04	0.3	0.02	0.6	0.05
8	1.5	0.12	1.0	0.08	0.7	0.06	1.1	0.09
16	2.8	0.23	2.1	0.17	1.5	0.12	2.2	0.18
32	5.7	0.47	4.0	0.33	3.0	0.24	4.5	0.37
64	10.5	0.86	7.8	0.64	5.6	0.46	8.7	0.71
128	16.7	1.37	12.8	1.04	9.7	0.79	15.0	1.23
256	24.9	2.04	20.8	1.70	16.6	1.36	25.0	2.05
512	33.8	2.77	30.0	2.46	22.9	1.87	38.1	3.12
1024	37.8	3.10	55.6	4.56	27.5	2.25	77.3	6.33
2048	47.4	3.88	74.1	6.07	27.5	2.25	93.0	7.62

From Table 1 one can see that only a fraction of the underlying bandwidth of the Ethernet is used. This means that the likelihood of saturating the network is very low, even with many virtual nodes communicating at the same time.

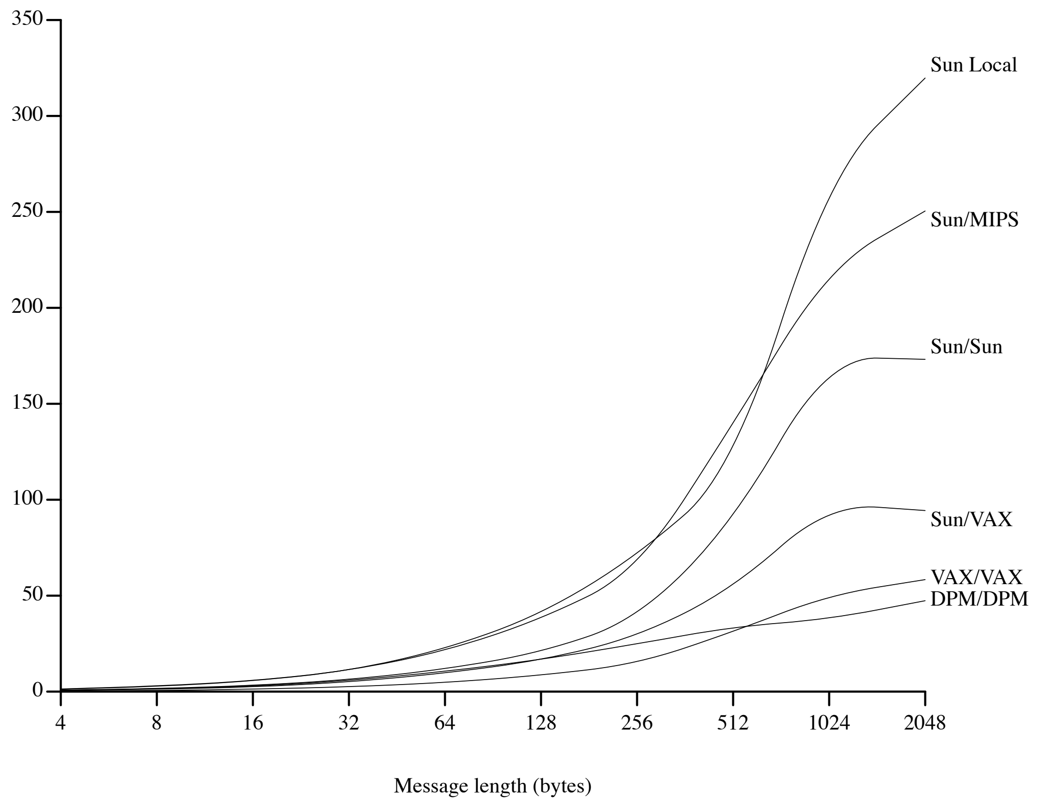


Figure 6. Timing for typical messages

For a typical message size of 32 bytes say, the VTM could accommodate many pairs of communicating nodes. Since each pair is only using 0.2-0.4 percent of the Ethernet bandwidth at least 200 such pairs could use the network simultaneously without saturation. However, if needed, higher speed communications could be accomplished by simplifying the communication protocols thus reducing the software overhead

and bring the effective speed closer to the bandwidth of the physical network.

Figure 6 shows a portion of the timing information in more detail. For short messages the fastest communication is between two of the PTM processors (standalone with no operating system overhead). This shows that the dominating cost of the communications is the cpu time spent in the protocol supports since on these processors there is no competition for the cpu time. This also explains the poor speed of the VAX to VAX communications. On the VAX'es there was never more than approximately 30% of the cpu time devoted each communicating node, due to contention from other users. The timing relationships between the processors changes for longer messages (not shown in Figure 6). This is due to the different sizes of the message buffers. The page-size is 2K bytes on the Suns and there the message buffers are allocated in increments of this size. On the VAXes the message buffers are 1K and on the standalone PTM processors the message buffers are currently only 256 bytes. The dashed horizontal line represents effective speed of 9600 bits/sec and is included for comparison.

6. Debugging

Debugging parallel programs in a distributed environment is more difficult than sequential programs running on a conventional machine. The primary source of this added difficulty is the asynchronous sharing of information in the distributed environment. This sharing (via message passing) between processors with different clocks introduces a time-dependence into the distributed program. The execution characteristics of the program are no longer solely decided by its inputs, but are influenced unpredictably by interactions between autonomous processors, the physical characteristics of the communication medium and by the behavior of other programs sharing these resources. Bugs manifest themselves sporadically and often are not reproducible. Programs can no longer be instrumented to collect information on their execution environment, because this now changes their timing characteristics and thus their behavior.

A typical development cycle of an application in the VTM environment involves first designing and testing the code with the whole virtual machine residing on one physical processor. This eases the task of monitoring and keeping track of output from all nodes, and eliminates most of the timing dependencies mentioned above since the communication is now all driven by the same clock. The code may be instrumented for debugging without changing its execution behavior. Once the program runs bug-free on one clock, it can be distributed over several physical processors. Any anomalous behavior that is now detected must be caused by timing problems. This change from a single clock to a truly distributed execution may not involve any recompilation or relinking of the code, but simply a change to the configuration file describing the mapping of the virtual machine.

Problems with timing must still be found and corrected, and for the reasons mentioned above, this must be done with minimal effect on the timing characteristics. One way to do this is to dedicate a separate processor to the task of monitoring all processes. This processor can be programmed to condense and abstract information from the other processors in the system, and prepare it for human consumption. This is done by a "monitor-server" residing on a processor with a graphical display. The user has complete control over the information that is sent to the monitor as well as how this information is interpreted and presented. In essence, users write their own monitors using the primitives provided. An example of such monitoring display is shown in Figure 7. It was designed to keep track of the execution of one of the chess programs that have been implemented using a VTM. In the lower left corner, the current configuration is displayed and the horizontal bars represent work completed by the named nodes. The gaps in these bars represent synchronization points between iterations of the progressive deepening search. The numbers beside the bars measure how many nodes are searched by each processor during the previous iteration. The rest of the display is then used by the application itself (in this case ParaBelle [11]). The use of such visual representation of the execution and communication characteristics of distributed programs provides a more intuitive understanding of the behavior of parallel algorithms, an understanding that is difficult to obtain simply by analyzing the results.

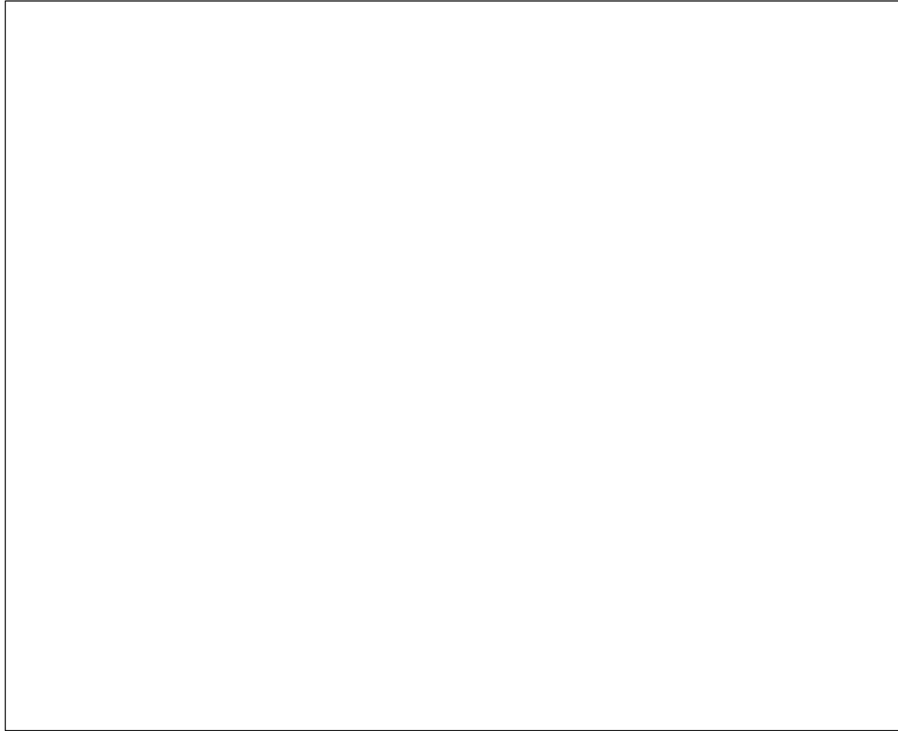


Figure 7. Example of VTM monitor display

One technique that has proven useful, is to design timing discrepancy tolerance into the algorithms. One example of this is a uniform message format. A node, expecting a message of a particular type, may receive a message of an unexpected type because of delays or other timing-related problems. If all messages are tagged, the receiving node can determine what action to take upon receiving the unexpected message. An example is a message about a piece of work already completed. The parent say, has not yet noticed that a child has completed its work and sends it some additional information. The child is waiting for more work, and if the message is tagged it will simply be discarded as opposed to being interpreted as new work.

Polling is another method that should be considered, especially as an alternative to interrupt-driven code. In the VTM environment polling is used to eliminate the danger of deadlock because of a lost interrupt. On an interrupt polling must be used to determine from which of the children (or the parent) the message originated. When this is done all communication paths are polled and all outstanding messages read. This eliminates the danger of deadlock should interrupts be lost when two or more messages arrive simultaneously. In some applications, polling can replace interrupts, because polling can be made less expensive, since no state-change or context-switch involved. However, one must poll often enough to minimize communication delays, and yet not so often that excessive time is spent on the polling function.

7. Applications

The facilities described in this report have been used primarily in experiments with parallel tree-searching algorithms. One vehicle for these experiments are two chess programs, Parabelle. [11] and Para-Phoenix [12]

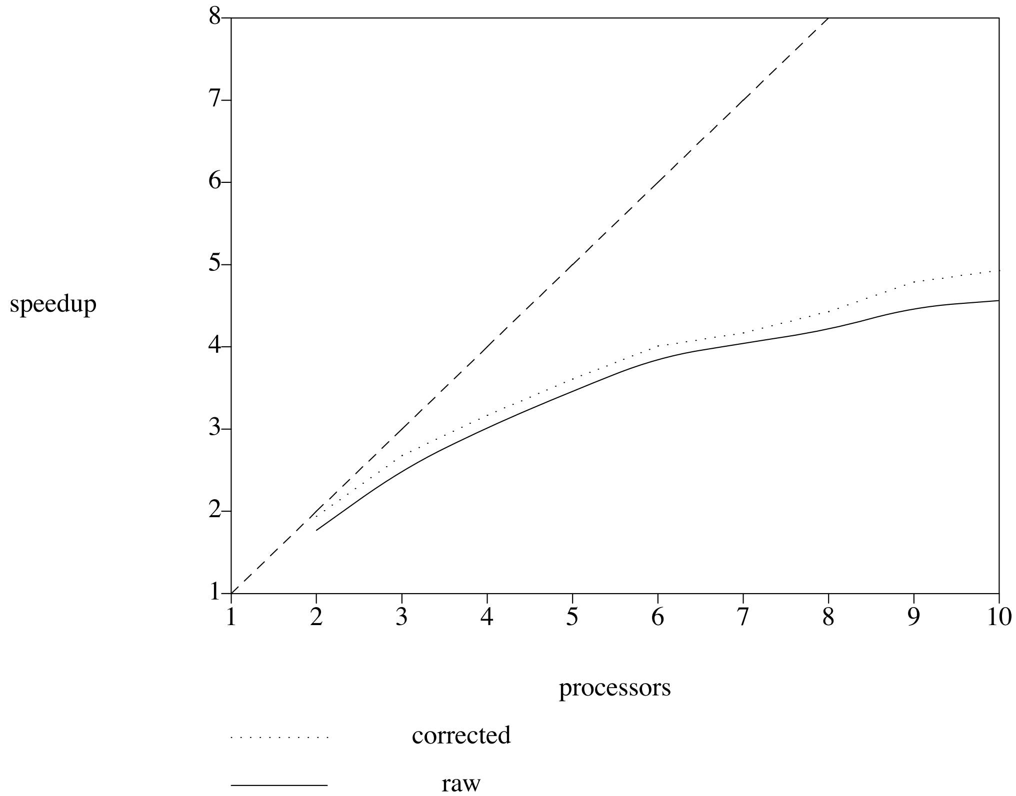


Figure 8. ParaBelle Speedup

Parabelle was used to explore the effect of using local and global memory to share information about the subtrees seen by different processors. Such information sharing can reduce the search times substantially. With tree machines it is common that one processor has far more memory than the others, and so is used to hold shared information. However, considerable processing time may be lost when several processors must await access to global tables. Conversely, local tables may become overloaded during a search, and so lose their effectiveness. The forerunner of our present VTM system was used to explore the tradeoffs in local/global memory usage. [13] Parabelle itself consisted of a processor tree of depth 1 and fanout f. Thus the trees were searched in a special way, using the PVsplit algorithm, [13] with f processors. One of these processors was called the master and had extra duties, such as allocating work to itself and other processors, and polling them at convenient intervals for their results. Parabelle has now been implemented as a VTM and is currently being used to explore the power of different processor tree configurations (depth and fanout) to determine what control over the synchronization losses may be possible by this means. The speedup achieved for this implementation of Parabelle is shown in Figure 8. The dotted line represents times that have been adjusted for the fact that the UNIX-based machines run approximately 20% slower than the standalone 68000's used in the previously mentioned processor tree machine. The tree configurations used to produce these results for ParaBelle where run on the 6 processor PTM, except for the root process which always runs on a UNIX-based machine. Hence, for the 8 through 10 processor cases UNIX-based machines where added.

ParaPhoenix, on the other hand, was the first major VTM application. It used the same search tree splitting algorithm and processor tree architecture as Parabelle, but a separate process was named the master. Since the master only manages the other processors it had ample time to measure their activity and effective CPU speed. Thus ParaPhoenix was used to measure accurately the synchronization losses of the system, and to identify the serious nature of this overhead. [12] Even so the master had little to do, so the VTM facility was used to allocate a tree-search process to that machine.

Other applications include a parallel implementation of the branch-and-bound algorithm for the traveling salesman problem, which is being used to investigate the tradeoff between communication overhead and synchronization overhead. In the planning stage is real time animation application. [14] The VTM facilities have also been used for teaching purposes, specifically in parallel processing and operating systems courses.

These experiments attempt to measure experimentally some of the costs and overheads involved in distributed processing. Theoretical investigations into parallel algorithms rarely take into account the losses attributed to communications or synchronization overhead. This is understandable, since they are difficult to formulate in the theoretical model of the computation. It is therefore important to have access to facilities to measure these and other poorly understood aspects of parallel algorithms.

8. Conclusions

With the proliferation of low-cost but powerful processing elements it becomes increasingly important to address the question of how to best deploy many such processors in a single system. There is no one correct method of doing so. It is necessary to evaluate different alternatives, and facilities must exist to experiment with different algorithms and different programming techniques. While it is relatively easy to build distributed systems hardware, it is difficult to program and use such a system. This difficulty is often compounded in research systems by both the lack of operating system support for the design and development phase, and the lack of run time support.

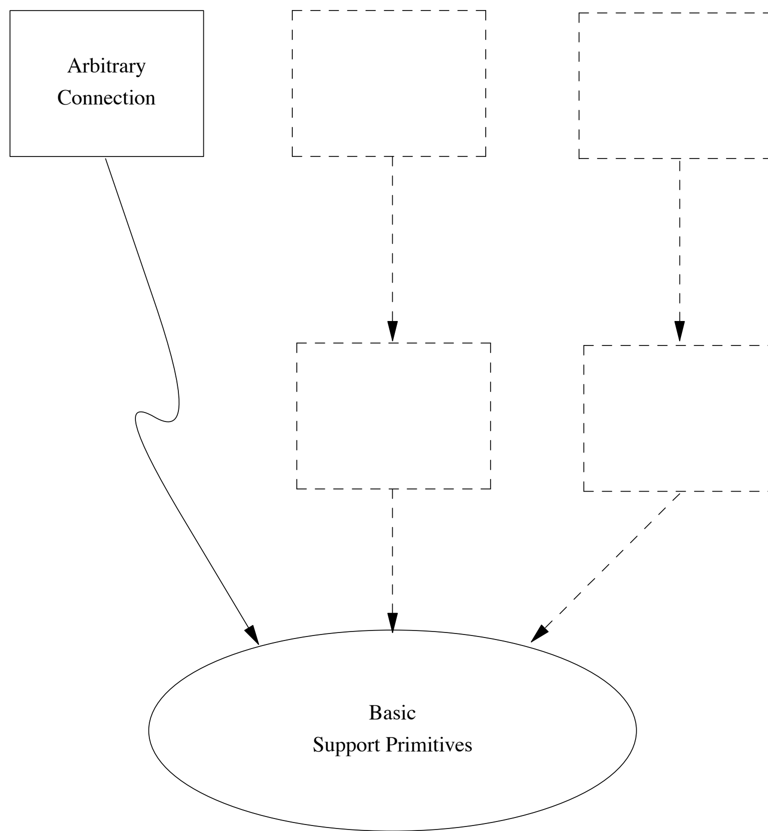
The facilities described here make it possible to develop and test distributed algorithms under near normal conditions. As long as the results are interpreted correctly, virtual machine architectures can provide valuable insight into the behavior of non-existing, new, or unavailable real machines. [3] Algorithms for execution on these architectures can be developed, tested and debugged using this facility. While the primary purpose of the VTM architecture is to apply several processors to a single application, it can also be used to model large multi-processor systems and study their processor synchronization and communication delay properties.

Future plans for expanding this facility include providing virtual environments for interconnection methods other than a tree (such as hyper-cubes [10] and simple bus structures), and providing simpler and faster communication protocols, thus making the virtual environment competitive with tightly coupled systems, while retaining all the advantages of operating system support procedures.

Acknowledgements: The hardware support by Steve Sutphen and the UNIX networking software support from Dick Foster is gratefully appreciated. Jonathan Schaeffer, Steve Sutphen and Alexander Reinefeld provided constructive criticism on the earlier drafts of this report. Financial support from the Natural Sciences and Engineering Research Council of Canada in the form of equipment grant E5722 and operating grant A7902 was vital to the success of this research project.

Appendix I
Basic Routines

The description of the routines used to implement the Virtual Tree Machine is divided into two parts. Here we describe the primitive routines that allow the definition of a virtual distributed machine with arbitrary interconnection topology. In Appendix II the Tree Machine primitives are described. These are implemented on top of the routines described here and restrict the interconnections to a tree-like structure. The reason for this two-level implementation is twofold. The VTM primitives and VTM configuration files are simpler as they deal only with tree-structured interconnections, they also allow the user to create pure recursive applications since, at each node in the tree, none of the primitives need know anything about portions of the tree other than its children at its parent. Secondly, this implementation scheme allows other restrictive interconnection to be implemented on top of the basic routines.



Appendix II
Virtual Tree Machine Support

The routines described here form the basis for the VTM implementation. In Appendix I we described the procedures on which these are based, so unavoidably there is some duplication of information.

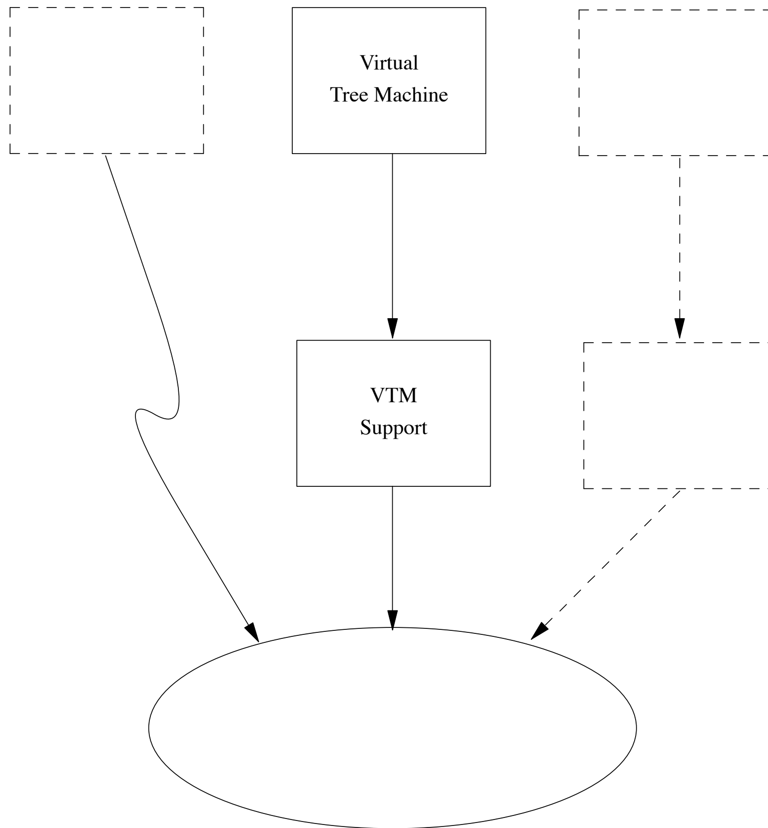


Table of Contents

1. Introduction 1
2. The Virtual Tree Machine 1
3. Implementation of Virtual Machines 4
3.1. Basic Support Routines 5
4. A VTM Example 6
5. Communication Speed 9
6. Debugging 11
7. Applications 12
8. Conclusions 14
References

References

1. F. W. Burton and M. Huntbach, ‘‘Virtual Tree Machines,’’ *IEEE Transactions on Computers* **C-33, 3**, 278-280 (1984).
2. S. A. Browning, ‘‘A Tree Machine,’’ *Lambda* **6**, 31-36 (1980).
3. Myrias 4000 System Description, Myrias Research Corporation, Edmonton, May 1984.
4. C. Lam, B. C. Desai, J. W. Atwood, S. Cabilio, P. Grogono and J. Opatrny, "A Multiprocessor Project for Combinatorial Computing," *CIPS Session 82*, Saskatoon, May 1982, 325-329.
5. T. A. Marsland and M. Campbell, ‘‘Parallel Search of Strongly Ordered Game Trees,’’ *Computing Surveys* **14**, 533-551 (1982).
6. G. Lindstrom, The Key Node Method: A Highly-Parallel Alpha-Beta Algorithm, Tech. Rep. UUCS 83-101, Dept. of Computer Science, Univ. of Utah, Salt Lake City, March 1983.
7. D. Ritchie and K. Thompson, ‘‘The UNIX Timesharing System,’’ *Comm. of the ACM* **17,7**, ?-? (1974).
8. B. W. Kernighan and D. M. Ritchie, The C Programming Language, Prentice Hall, 1978.
9. S. J. Leffler, R. S. Fabry and W. J. Joy, A 4.2BSD Interprocess Communication Primer (DRAFT), Computer System Research Group, Univ. of California, Berkeley, December 1983.
10. C. L. Seitz, ‘‘The Cosmic Cube,’’ *Communications of the ACM* **28**(1), 22-33 (January 1985).
11. T. A. Marsland and F. Popowich, "Parallel Game-tree Search," *to appear IEEE Transactions on PAMI*, May 1985.
12. J. Schaeffer, M. Olafsson and T. A. Marsland, Experiments in Distributed Tree-Search, Tech. Rep. 84-4, Computing Science Dept., Univ. of Alberta, Edmonton, June 1984.
13. F. Popowich and T. A. Marsland, Parabelle: Experience with a Parallel Chess Program, Tech. Rep. 83-7, Computing Science Dept., Univ. of Alberta, Edmonton, August 1983.
14. W. W. Armstrong and M. Green, "The Dynamics of Articulated Rigid Bodies for Purposes of Animation," *to appear Graphics Interface ’85*, 1985. 14

Appendix I
Basic Support Routines 15

Appendix II
Virtual Tree Machine Support 16