A UNIX® BASED VIRTUAL TREE MACHINE

Marius Olafsson
T.A. Marsland

Department of Computing Science
University of Alberta
Edmonton T6G 2H1

For presentation at the CIPS Conf., Montreal, 4 June 1985.

ABSTRACT

The paper 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.

RESUME

On décrit un environnment qui convient aux expériences du procédé réparti de la computation. Notre système offre au chercheur un moyen facile de préparer, d'executer et de vérifier des algorithmes parallèles, en fournissant les méchanismes pour décrire les configurations variées sur un réseau d'ordinateurs. Le système se laisse adapter notamment aux interconnections en forme d'arbre; en consequence nous appelons l'environment une "Virtual Tree Machine." (Le système est realisé sur un réseau d'ordinateurs VAX 11/780 et Sun-2). On démontre dans les examples l'utilisation de ces facilités pour développer des algorithmes parallèles et pour étudier l'architecture des réseaux d'ordinateurs.

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 [4]. Tree-structured topologies have been proposed to reduce the connections between processors in distributed systems [3]. 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 [1]. 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 [5] and tree-searching algorithms [7,8].

2. The Virtual Tree Machine

This paper 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, see Figure 1. In addition, the system includes six standalone Motorola 68000's, 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 restriction on their usage as a part of the VTM environment is that they do not support more than one process per physical processor.

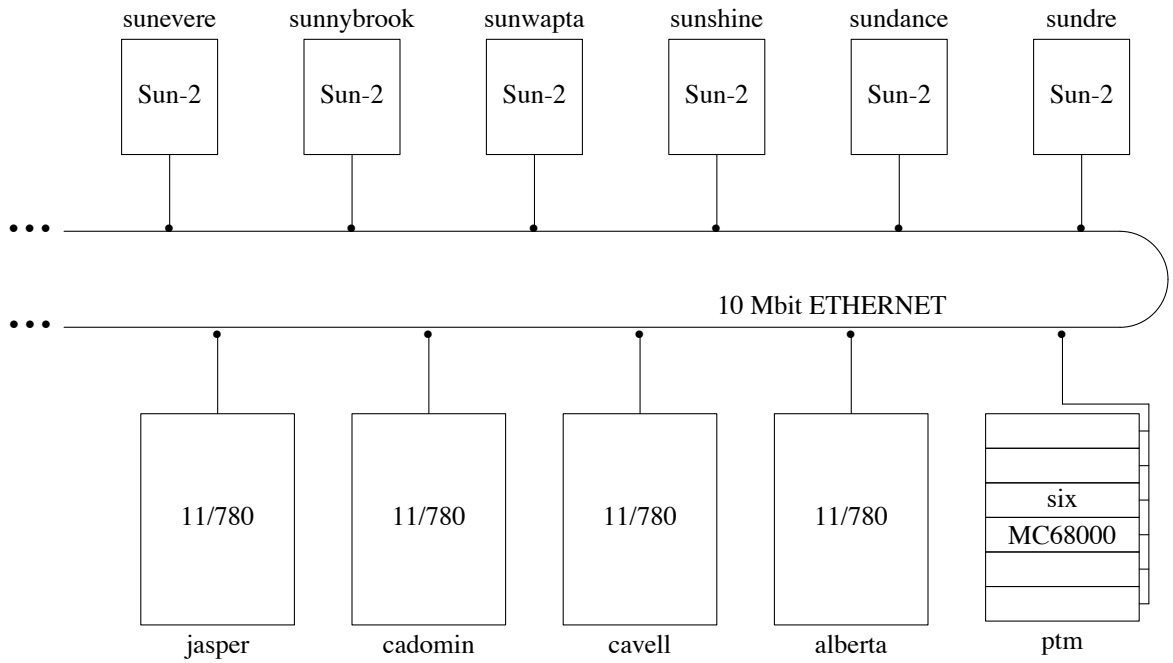


Figure 1. Computing Facilities

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 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. The most important of these routines are described below:

fanout = NodeInit(name, file)

which is used to give a name to the VTM and establish communications between a parent and its children. *NodeInit* creates descendant nodes, recursively from a high-level description of the desired interconnection topology received from the parent. In the root node, this description is read from a file named by the second parameter. Once all communication paths have been created, *NodeInit* returns control to the user's application. In each node, *fanout* specifies the number of children created. Later, a parent may send/receive message to/from its children via the following procedures:

Csend(child, message, length, interrupt, trace)
Creceive(child, message, length, trace)

Similarly a child communicates with its parent with the following:

Psend(message, length, interrupt, trace)
Preceive(message, length, trace)

Messages may be specified to interrupt 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. In addition, facilities exist for enabling and disabling interrupts, to poll descendants and the parent for outstanding messages and to add and delete nodes from the tree. More information on these routines is found in our report [10].

3. Implementation

The VTM environment is built on top of the UNIX networking primitives. These primitives allow processes to communicate via a variety of protocols and connection strategies [6]. The current implementation of the VTM uses reliable two-way communication channels (called stream-sockets). The semantics of the stream-sockets are similar to UNIX pipes, 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 VTM node-server so that no special administrative consideration needs to be given the VTM over other servers on a particular machine. Thus, the user views the VTM as a reconfigurable tree of virtual processors (arbitrary depth and fanout possible) each with a large amount of memory, running under the control of an operating system that provides access to various peripherals.

As an example, consider the creation of a VTM to execute the configuration of processes depicted in Figure 2.

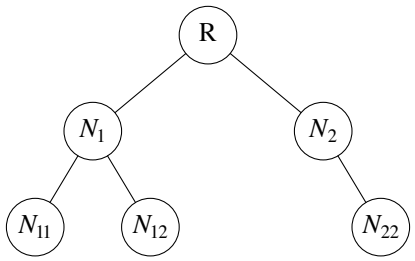


Figure 2. Process Tree

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

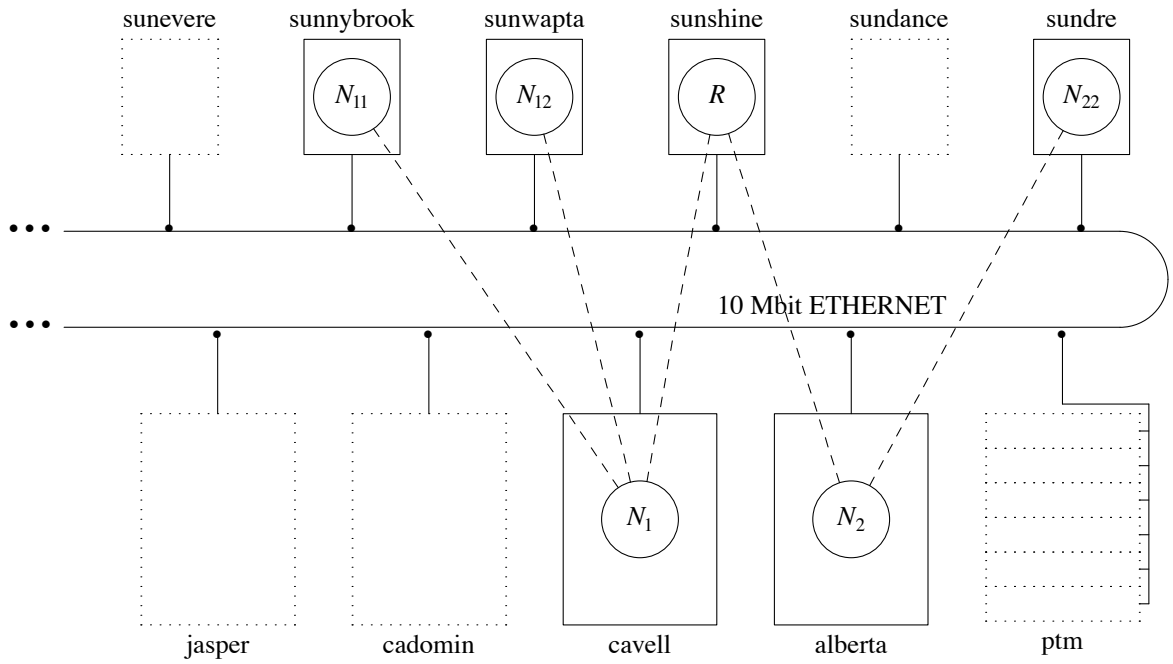
That is, the root resides on a SUN-2 processor, called *sunshine,* the interior nodes are on VAX-11/780 processors and the leaf nodes on SUN's. The mapping between the virtual machine and the physical hardware is described in *configuration file*. Each line in the configuration file represents a virtual node in the tree 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 2 to the available hardware.

```
sunshine;2;0
    cavell;2;0; node I;; out1; err1;
       sunnybrook;0;0; node L;; out11; err11;
       sunwapta;0;0; node L;; out12; err12;
    alberta;1;0; node I;; out2; err2;
       sundre;0;0; node L;; out22; err22;
```

When the root process on *sunshine* is started it sends a service request to the node-server on *cavell*. The server executes the file *node* (from the fourth field in the configuration entry for the

process on *cavell,* prepended with the users home directory), gives it the execution parameter "I" (indicating internal node), 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. 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, *NodeInit* returns and the application is ready to start work, since all communication paths have now been established. The VTM created is shown in Figure 3. 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.



------ Virtual communication paths

○ Node processes

Figure 3. Mapping of Processor Tree onto Hardware Configuration

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

```
fanout = NodeInit(name, cfile);
....
  connection has been made between
  the parent and 'fanout' children
....
Preceive(buf, n, TRACE);
....
  process data from parent and
  prepare to send on to children
....
for i from 1 to fanout
    ....
      prepare data for child # i
    ....
    Csend(i, buf, length, NOINTS, TRACE);
....
  carry out intended application
....
for i from 1 to fanout
    Creceive(i, buf, length, TRACE);
    ....
      process message from child # i
    ....
....
  prepare a reply to parent
....
Psend(buf, length, NOINTS, TRACE);
```

The process containing the above code segment is invoked by the node-server on its host machine. After invocation, *NodeInit* waits for the parent to send the configuration of its tree branch. Once received, *NodeInit* transmits requests to start this node's children (if any). When *NodeInit* returns, communication has been established with the parent (from which the node receives its work via *Preceive)* and its children (to which it sends some units of work via *Csend)*. When this node has finished its work, it receives the results from its descendants (via *Creceive)* and finally transmits its results to its parent (via *Psend)*. 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 *Preceive* has a corresponding *Csend* call in the parent node and every call to *Psend* corresponds to a *Creceive* call in the parent.

4. 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 "debug-server" residing on a processor with a graphical display. The user has complete control over the information that is sent to the debug-server as well as how this information is interpreted and presented. In essence, the users write their own debug-servers using the primitives provided [10]. 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.

Another 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 typed, 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 typed it will simply be discarded as opposed to being interpreted as new work.

Polling is one technique that should be considered 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, since polling can be made less expensive (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.

5. Applications

The facilities described in this paper have been used primarily in experiments with parallel tree-searching algorithms. The vehicle for these experiments are two chess programs, Parabelle [9] and ParaPhoenix [11].

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 [9]. 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 [9], 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.

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. Even so the master had little to do, so the VTM configuration also allocated 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 [2]. 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.

6. 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 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 [1]. 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 a hyper-cubes[12] 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 paper. 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.

References

1. Myrias 4000 System Description, Myrias Research Corporation, Edmonton, May 1984.
2. W. W. Armstrong and M. Green, "The Dynamics of Articulated Rigid Bodies for Purposes of Animation," *to appear Graphics Interface '85*, 1985.
3. S. A. Browning, "A Tree Machine," *Lambda* **6**, 31-36 (1980).
4. F. W. Burton and M. Huntbach, "Virtual Tree Machines," *IEEE Transactions on Computers* **C-33, 3**, 278-280 (1984).
5. 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.
6. 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.
7. 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.

8. T. A. Marsland and M. Campbell, ‘‘Parallel Search of Strongly Ordered Game Trees,’’ *Computing Surveys* **14**, 533-551 (1982).

9. T. A. Marsland and F. Popowich, "Parallel Game-tree Search," *to appear IEEE Transactions on PAMI*, May 1985.

10. M. Olafsson and T. A. Marsland, Implementation of Virtual Tree Machines, Technical Report (in preparation), Computing Science Dept., Univ. of Alberta, Edmonton.

11. 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.

12. C. L. Seitz, ‘‘The Cosmic Cube,’’ *Communications of the ACM* **28**(1), 22-33 (January 1985).