A Network Multi-Processor for Experiments in Parallelism

T.A. Marsland
T. Breitkreutz
S. Sutphen

Computing Science Department
University of Alberta
Edmonton
Canada T6G 2H1

Final draft for Concurrency: Practice and Experience
March 1991

ABSTRACT

Although both shared memory and loosely coupled parallel computing systems are now common, many still do not offer an easy way to design, implement, and test parallel algorithms. Our system provides software tools that make possible a variety of connection structures between processes. These structures are said to form a 'Network Multi-Processor', which is implemented on a local area network of heterogeneous UNIX-based timesharing computers, plus a set of processor boards dedicated to an application so that accurate timing measurements can be made. We explain how these tools have been used both to aid parallel algorithm development and to explore the properties of different computer interconnection methods.

March 31, 1991

A Network Multi-Processor for Experiments in Parallelism

T.A. Marsland
T. Breitkreutz
S. Sutphen

Computing Science Department
University of Alberta
Edmonton
Canada T6G 2H1

Final draft for Concurrency: Practice and Experience
March 1991

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, distributed systems. Closely coupled systems have traditionally been more popular since they can be used directly to speed existing algorithms and programs. For example, powerful vector processors are now well established and most contemporary systems use some degree of pipelining, but have the disadvantage that applications must be specially tailored to the hardware design. Also they are only well-suited to certain classes of problems. Those systems provide what is often referred to as 'fine-grained' parallelism. This paper deals primarily with 'large-grained' parallelism.

Until recently, progress in experimental computer science was slowed by the cost and special purpose nature of the equipment. Specifically, in early distributed systems researchers managed with a collection of connected processors, each with little or no I/O capability, rudimentary operating system support and a small memory[1]. On such sys-

tems experiment management was often difficult and the lack of flexibility restricted experiment design. With the widespread use of local area networks, experimenters have taken advantage of existing computing facilities, have drawn on the services of a powerful operating system (with such capabilities as virtual memory management) at each node, and have designed their distributed algorithms in high-level programming languages. Often researchers had to modify existing operating systems to accommodate the remote processes or communication schemes, for instance the U* system runs on NEST — a modified UNIX† kernel[2]. 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[3, 4]. Naturally, running under an operating system 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 of the more powerful tool set.

A problem faced by designers of all parallel processing systems is the tradeoff between communication speed and the complexity of the connection structure[5]. Tree-structured[6] and hypercube[7] topologies have been proposed and constructed to reduce the connections between processors in distributed systems. The advantage of a tree structure is that the number of links only increases linearly with the number of processors, thus making possible the construction of systems with thousands of processors without a prohibitively expensive interconnection network[8]. An advantage of both topologies is that some problems map naturally into them. These include some NP-complete problems, such as combinatorial methods requiring exhaustive search[9], and tree-searching

---

† UNIX is a trademark of Bell Laboratories.

algorithms[10, 11].

2. The Network Multi-Processor

Here we describe 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 Network Multi-Processor (NMP). It is installed on a local area network and supports a heterogeneous collection of machines from a variety of vendors such as Sun Microsystems, MIPS, Silicon Graphics and Digital Equipment. Each machine has an operating system whose network primitives are equivalent to the 4.3BSD-based version of UNIX [12, 13].

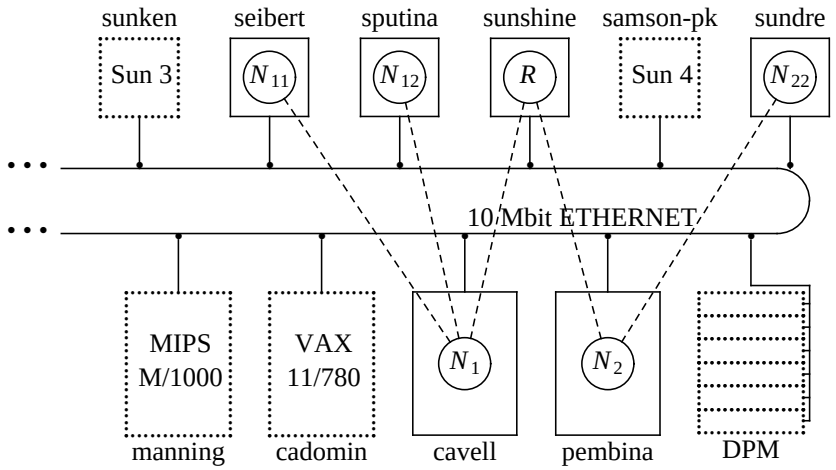


Figure 1. Mapping of processor tree onto selected hardware

Figure 1 shows part of our facilities. It includes some dedicated Motorola 68000's, with restricted operating system support, for use when critical timing measurements must be made. Since these processors support the activities of only one application process

they are collectively referred to as the dedicated processor machine (DPM). Most of the standard UNIX programming environment is available in the DPM through the compile-time library, as are networking support, timers, and the standard I/O library; but file I/O and process management are not provided. The experimenter views the system as a collection of processing elements — each with adequate memory, disks and other I/O devices, and with arbitrary communication paths to other processing elements. In reality, NMP 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 the node processes of the NMP either according to the description provided by the user, or automatically based on ‘least-busy’ computers. During the early development the whole application might reside on one physical processor, only later being distributed over the selected machines for production use.

The interface to the NMP is a collection of user procedures, written in the C language[14] and callable from application programs. These procedures handle connection establishment, connection initialization, exchange of messages and interrupt handling, as well as providing information on the configuration and layout of the NMP being used.

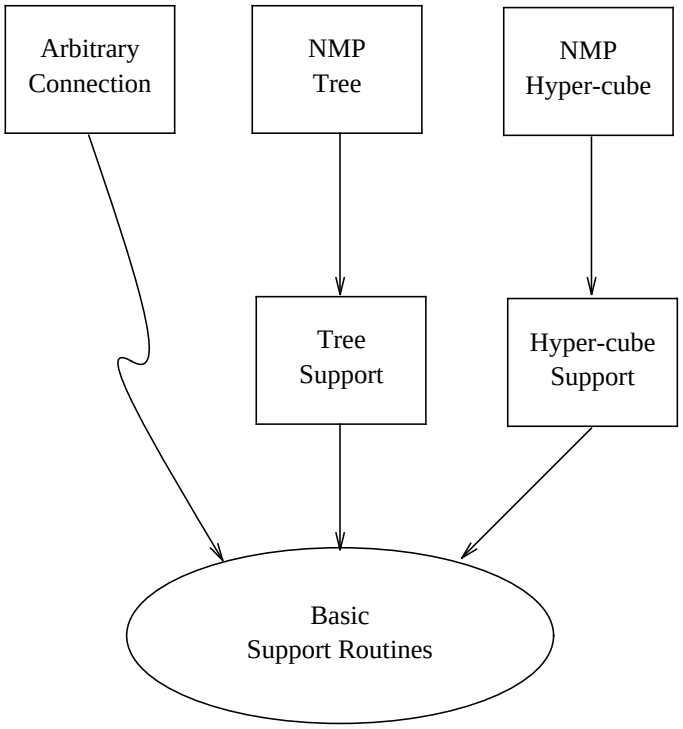


Figure 2. Hierarchy of network multiprocessor implementation

2.1. Implementation of a Network Multi-Processor

The programming environment is implemented using a set of standard NMP routines that allow the creation of virtual multiprocessors with arbitrary interconnection structures, shown in Figure 2 as the 'Basic Support Routines'. These routines are in turn implemented with the Berkeley interprocess communication networking primitives that allow processes to communicate with a variety of protocols and connection strategies[15]. The current implementation of the support routines uses reliable two-way communication channels, called internet-domain stream sockets, which are similar to UNIX pipes[12], except that the communicating processes need not reside on the same physical machine. Pipes were used as the communication mechanism in our earlier work

on distributed heterogeneous systems[16]. There are two main aspects of the Berkeley networking primitives that make them a good basis for implementing a virtual processor system. First, the socket interprocess communication model is internally consistent; communication between processes is indistinguishable from communication between processors. That is, communicating 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 node server so that it needs no special administrative consideration over other servers on a particular machine.

The 'Basic Support Routines' provide a completely general interconnection structure, any node can be connected to any other. The placement of the nodes on specific machines, and the interconnecting vertices are specified at run time in a 'configuration file' (described later). Many problems are well-suited to the use of tree or cube architectures. To reduce the tedium and risk of making mistakes in these more structured applications, special tree and cube support routines overlay the basic support and so provide a more natural environment for applications that conform to those two regular structures[17].

2.2. Standard NMP Routines

The execution environment consists of a collection of virtual processors (UNIX processes) whose stream-socket connections over an Ethernet are created during initialization. To get started, a node (process) initialization function is invoked, as follows:

```
neighbors = NodeInit (NodeType, ConfigFile);
```

There are two types of node initializations. The first, `NodeType=ROOT`, is used in the node that interacts with the user. This 'control node' initializes itself by reading a `ConfigFile` and creating other nodes (processes) as specified in that file. All other nodes,

with `NodeType=INTERNAL`, receive the configuration file information from their creator; that is, the `ConfigFile` parameter is ignored.

The configuration file consists of two parts: an ordered set of descriptor lines, one per node, naming the host process, execution file name and so on; followed by a connection matrix, showing how the ordered nodes are connected. A descriptor line has the form:

```
HostName; Bits; FileName; Sin; Sout; Serr;
```

where `HostName` is the physical host on which the node is to run;

`Bits` is an integer whose bit values specify such things as the desired level of debugging support, and whether the host is a DPM.

`FileName` identifies the code to execute on that node;

The three last fields contain the names of files to be opened as the node's standard input, standard output, and standard error streams respectively.

The second part of the initialization file is the connection specification. It is a lower triangular matrix of 0's and 1's, where a 1 in row *N* and column *M* represents the desire for bi-directional communication between nodes *N* and *M*. `NodeInit` creates the remaining nodes from the interconnection description received from its creator. Since the user defined communication paths are used by NMP to 'bootstrap' itself, any unconnected portions of the configuration will be connected to the root automatically. Once all communication paths have been established, control returns to the user's application.

At each node, `neighbors` measures the number of adjacent (connected) nodes. For identification and management, each node in the NMP system is assigned a unique integer identifier, `NodeId`. A small set of functions provide a node with information about itself and its neighbors. For example, `GetMyId()` provides the `NodeId` of a node, `GetNodes()` returns the number of nodes configured in the current NMP, and

`IsConnected(NodeId)` returns non-zero if the node given in the parameter is connected to the calling node.

Once it is known that a pair of nodes is connected, they may communicate with each other via the following functions:

```
len = SendNode (NodeId, Message, Length);
len = RecvNode (NodeId, Message, Length);
```

where `NodeId` identifies the source or destination, and `Message` is the address of a buffer containing the bytes to be sent, or a space into which the received message will be put. These functions return the number of bytes transmitted or received. Note that `RecvNode` waits until all `length` bytes have arrived from the specified node, so a polling capability is also provided to identify nodes with outstanding messages.

Facilities also exist for managing abstractions of the Berkeley signal mechanism[13]. These include routines to enable, disable, hold and release signals, and to specify the signal handler, as follows: `EnableInt()`, `DisableInt()`, `HoldInt()`, `ReleaseInt()`, and `SetHandler(Handler)`, where `Handler` is the address of an interrupt handling procedure that is invoked whenever a message arrives. Interrupts are not generated if no handler has been set, or if `SetHandler` is called with a null parameter.

Although interrupt driven message management can work well, they are unstructured, and there are always cases when interrupts may be lost (e.g. occur nearly simultaneously) leading to extended waits for responses. One way to protect against such a problem is for a process to poll others for pending messages. For this reason, two routines are provided to check for messages from neighboring nodes (all neighbors or a particular one):

```
count = PollAll (Nodes[], Block);
count = PollNode (NodeId, Block);
```

In the case of `PollAll`, the id's of nodes that have messages outstanding are returned in the array parameter, `Nodes`. These routines count the nodes with messages pending, returning 0 if no messages are outstanding. Setting `Block` to `TRUE` instructs the routine to wait until a message arrives.

2.3. Dynamic Reconfiguration

There are cases where one would like to change the size or shape of the NMP after the application has started. This could be useful for load-balancing, e.g., deleting a node on an overloaded machine and recreating it on a machine with a lighter load. Similarly, if a node fails for some reason, a new node can be created in its place thus increasing the fault-tolerance of the system. These dynamic changes are made by using the following:

```
AddNode (HostName, FileName, Sin, Sout, Serr, Bits);
AddConnect (Type, NodeId);
```

where `HostName` is the name of the physical machine on which the new neighbor is to reside, and `FileName` is the executable file of the new process. The `Sin`, `Sout`, and `Serr`, character pointer parameters specify the files that will be opened as the new node's standard I/O streams. If a null string is supplied then `/dev/null` is opened. `Bits` specifies the debug flags and socket type, and `AddConnect` creates a connection between two existing nodes. A new connection must be requested by both nodes — one side with parameter `Type=ACTIVE`, the other with `Type=PASSIVE`. 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.

2.4. Tree Machines and Hypercubes

The basic support routines described above are capable of implementing any arbitrary interconnection, including more regular structures like trees and cubes. Initially our distributed computing research was limited to tree structured architectures and the NMP system was then called the *Virtual Tree Machine* (VTM)[18]. To support this regular architecture several primitives — similar to the ones for the basic support — were implemented. For example, since the connection structure is implicit, the `TreeInit` function reads a simpler configuration file that consists of processor description lines only. Routines `TreeRecvParent` and `TreeRecvChild` serve the function of `RecvNode`, while `TreeSendParent` and `TreeSendChild` are similar to `SendNode`.

Similarly, `CubeInit`, which models a symmetric hypercube processor environment[7], also manages without a connection matrix. Instead, the node descriptor lines are preceded by an integer that specifies the dimension of the cube desired. The specialized support includes several routines tuned for use in a hypercube environment. Modelling of small hypercubes is straightforward, and our standard distribution contains an example solution of the N-body body problem to illustrate use of the hypercube primitives.

3. Creating a Processor Tree

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

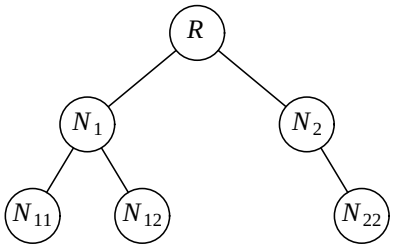


Figure 3. Process tree

First this configuration must be mapped onto the hardware. This can, and often is, done automatically by first noting the workload of the available computers and assigning the N least busy to the current application. There are no unusual restrictions on the number of nodes that can exist on one processor, but for clarity here we map them, one per physical machine as follows:

R on sunshine, N_1 on cavell, N_2 on pembina,
N_{11} on sputina, N_{12} on seibert, and N_{22} on sundre.

Here, the root resides on a Sun processor, called *sunshine*, the interior nodes are on VAX-11/780 processors (*cavell* and *pembina*) and the leaf nodes are on other Sun workstations, as Figure 1 shows. The mapping between the virtual machine and the physical hardware is specified in the simplified 'tree' form of the configuration file, shown in Figure 4.

```
sunshine; 2; 0
     cavell; 2; 0; branch -p1; ; out1; err1;
          seibert; 0; 0; leaf -p11; ; out11; err11;
          sputina; 0; 0; leaf -p12; ; out12; err12;
     pembina; 1; 0; branch -p2; ; out2; err2;
          sundre; 0; 0; leaf -p22; ; out22; err22;
```

Figure 4. Sample NMP-tree configuration file

Each line in the tree machine configuration represents a node and contains seven fields separated by semicolons. These fields are identical to the six fields of a 'Standard

Configuration File', described in section 2.2, except that there is an extra field—the second field—specifying the number of descendants of the node. Here we assume the general case in which incompatible processors have different home directories for the user. In a network filesystem with identical home directories on incompatible processors (for example, a Sun workstation and a VAX), the named files must still hold the correct type of executable for the corresponding processor in the configuration. Note that the default (`/dev/null`) is used for the omitted standard input.

When the root process is invoked by the user on *sunshine*, it calls `TreeInit` to read the contents of Figure 4 and translate it to the standard configuration file format, which in turn is written to a temporary file. That file is read by `NodeInit` (which is an embedded call in `TreeInit`) on *sunshine*, which then creates the specified virtual machine and sends a service request to the node server on *cavell*. When *cavell* receives the request it executes the file *branch* (the third field in the configuration entry for the node on *cavell*), with the execution parameters specified (here `-p1`), and returns to listen for additional service requests. The *branch* process on *cavell* receives the configuration from *sunshine* and notes that it has two children. It therefore transmits two requests, one to the server on *seibert* and another to the server on *sputina*. Both nodes deduce that they have no children and so respond that they successfully started their `leaf` process. The interior node on *cavell* then tells the root that all went well, and so the root knows that the left branch is complete. Meanwhile, the right branch has been started in parallel by the root transmitting a request to the node-server on *pembina*. Finally, `TreeInit` returns and the application is ready to start work, since all communication paths have now been established. Note that for many applications the `branch` and `leaf` processes will be identical. They are given different names here not only for clarity, but to point out that a

unique program can be executed at each node, if desired. The NMP connections created by Figure 4 are shown in Figure 1. Examples showing how to develop other applications are given in our local report[17].

3.1. Interprocess Communication

In a typical distributed computing system of UNIX-based computers on a local area network, several different NMP experiments can be performed at the same time. Here the Ethernet serves as a shared communication path, and processes from different applications may share the same processor.

To illustrate the communication and connection establishment features provided in the NMP environment, a skeletal C-code segment from an arbitrary interior node is presented in Figure 5. The process containing the code segment is invoked by the node-server on its host machine. After invocation, `TreeInit` waits for the parent to send the configuration of its subtree, and transmits requests to start its children (if any). When `TreeInit` returns, communication has been established with the parent (from which the node receives its work via `TreeRecvParent`) and its children (to which it sends some units of work via `TreeSendChild`). When the interior node has finished its work, it receives the results from its descendants (via `TreeRecvChild`) and finally transmits its results to the parent (via `TreeSendParent`).

```
# include <nmp/nmpdefs.h>
int i, n, fanout, length, len;
char *buf;
  .
  .
fanout = TreeInit (INTERNAL);      /* receive ConFig from parent */
                /* connections to 'fanout' children established */
len = TreeRecvParent (buf, n);            /* receive from parent */

for(i=1; i<=fanout; i++) {
    TreeSendChild (i, buf, length);          /* send to children */
}
  .
  .
for(i=1; i<=fanout; i++) {
    TreeRecvChild (i, buf, length);     /* receive from children */
}
TreeSendParent (buf, length);                /* reply to parent */
  .
  .
```

Figure 5. Sample communications code

The code excerpt of Figure 5 is identical on all nodes in the NMP (except the root where communication with the parent would be replaced with user interaction). Thus, in general, every call to `TreeRecvParent` has a corresponding `TreeSendChild` call in its parent node, and every call to `TreeSendParent` corresponds to a `TreeRecvChild` call in its parent.

4. Communication Aspects

One disadvantage of loosely coupled message-passing systems of this type is that communication may be slow and adversely affect the overall processing speed. For truly CPU bound applications that interact infrequently this would not be the case, and NMP accommodates that type of 'large grained' parallelism especially well. Fox[19] found that many problems fall into this class and can be mapped onto a system with modest

routing overheads. NMP is usually layered over the TCP protocol, although the `Bits` field in the configuration can specify that UDP be used.[1] The UDP properties fit well with some communication requirements like status monitors where packet loss and ordering are not critical. Alternatively one could add a layer of software to make the datagrams reliable as was done in PVM[21].

Figure 6 shows the effective communication times using the NMP primitives for the reliable, robust, TCP protocol. These results were obtained by measuring the average transit times for messages of varying lengths, up to 2 Kbytes. By way of interpretation, *DPM/DPM* represents two different processor boards (in the DPM) exchanging messages. *Sun Local* means two processes on the same Sun-3 workstation exchanging messages (i.e., the messages do not leave the machine but do go through the software layers down to the network interface and then loop back through the software layers to the other process). *VAX/VAX* represents communication between two independent VAX machines. Finally, *Sun/MIPS* represents Sun-3 and M/1000 communication, and so on. These results were averaged over several trials to factor out any transient conditions on the network.

As expected, longer messages yield near linear results (as the log scale of Figure 6 shows), up to the system buffer size or the underlying packet size of 1500 bytes. Beyond that the degradation of performance is of no surprise. The timing results from PVM[21] follow a similar pattern, although UDP was used there.

From Figure 6 one can see that only a fraction of the underlying 1.2 Mbytes/sec bandwidth of the Ethernet can be used, given the current software/hardware. This means

[1] TCP is a reliable, connection-oriented, stream protocol in the Internet Protocol suite, while UDP is a connectionless, unreliable, datagram protocol[20].

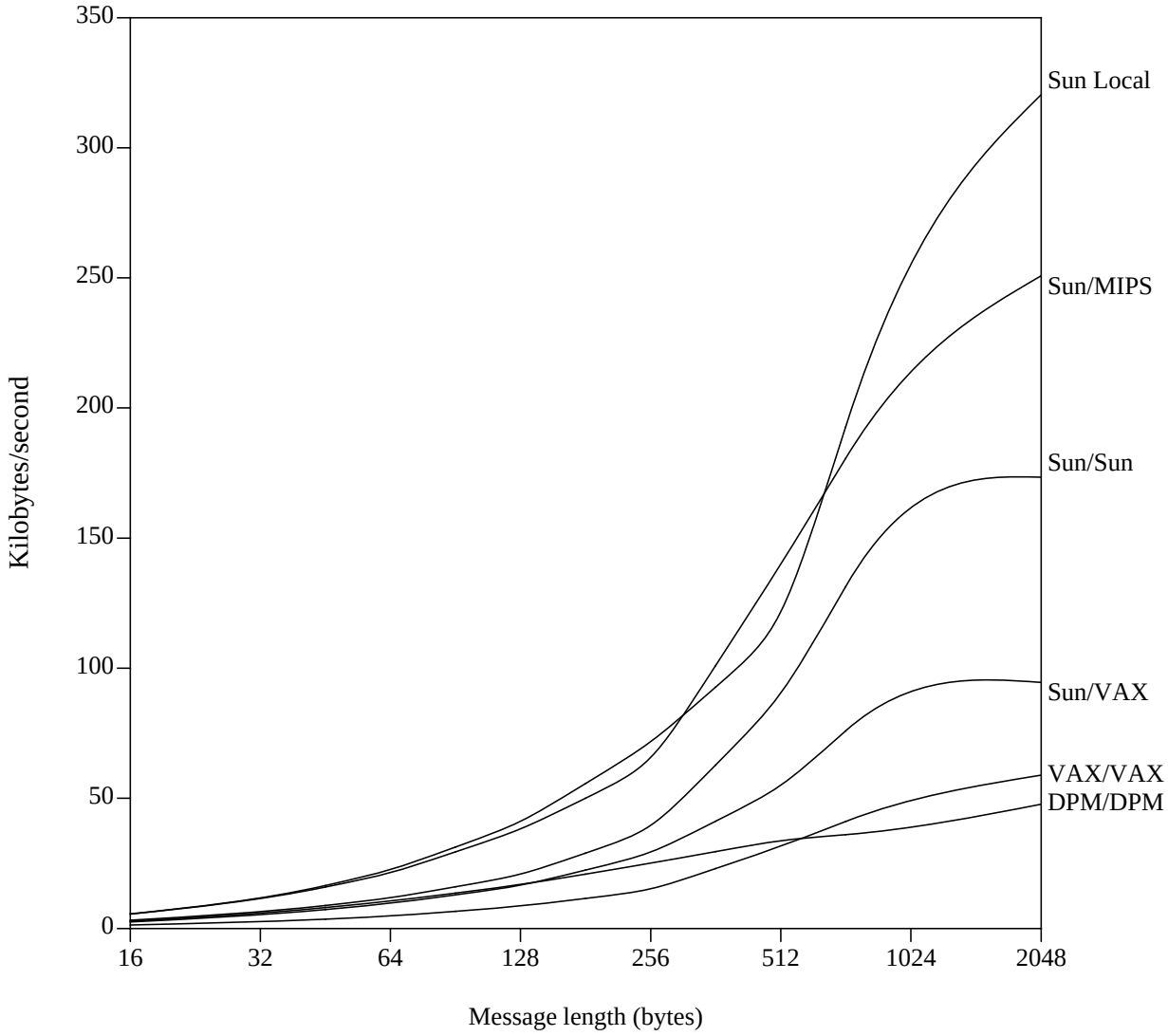


Figure 6. Timing for typical messages

that the likelihood of saturating the network is low, even when many nodes communicate at the same time. For a typical message size of 32 bytes say, NMP could accommodate many pairs of communicating nodes. Since each pair uses less than one percent of the Ethernet bandwidth about 100 such pairs could use the network simultaneously without saturation. In non-vital statistics-logging applications a higher communication rate is possible simply by requesting the cheaper UDP protocol. This reduces the software

overhead and brings the effective speed closer to the bandwidth of the physical network. For short messages (less than 256 bytes) the fastest communications occur between two processes local to the same Sun workstation. Our experience over the years has been that steadily improving communication protocols and faster processors have led to dramatic improvements in performance, as illustrated by the poor results for the older technology in the DPM/DPM, and the superior performance shown by the MIPS machines. Thus, the dominant cost of the communications is the CPU time spent in the protocol support. This also explains the poor VAX communications, where both a slower CPU and contention from other timesharing users hurt the communication rates. Figure 6 also shows how the transmission rate is affected by the different sizes of the message buffers. On the Suns, the message buffers are allocated in increments of the page size (2K bytes), but for the VAX computers the buffers are 1K bytes. On the DPM boards the message buffers are only 256 bytes, because their total memory is smaller. Even in the worst case with the smallest messages of 4 bytes, the transmission time is comparable to the capacity of a dedicated 9600 bit/sec channel.

5. Program Development

Asynchronous[19] parallel programs in a distributed environment are more difficult to debug than sequential programs running on a conventional machine[22]. More structured applications (Fox calls them *loosely synchronous* or *embarrassingly parallel* [19]) don’t exhibit this problem, because they are temporally better behaved. The primary source of this added difficulty is the asynchronous sharing of information in the distributed environment. This sharing (in our case via message passing) between processors with different clocks introduces a time-dependence into the distributed program. The

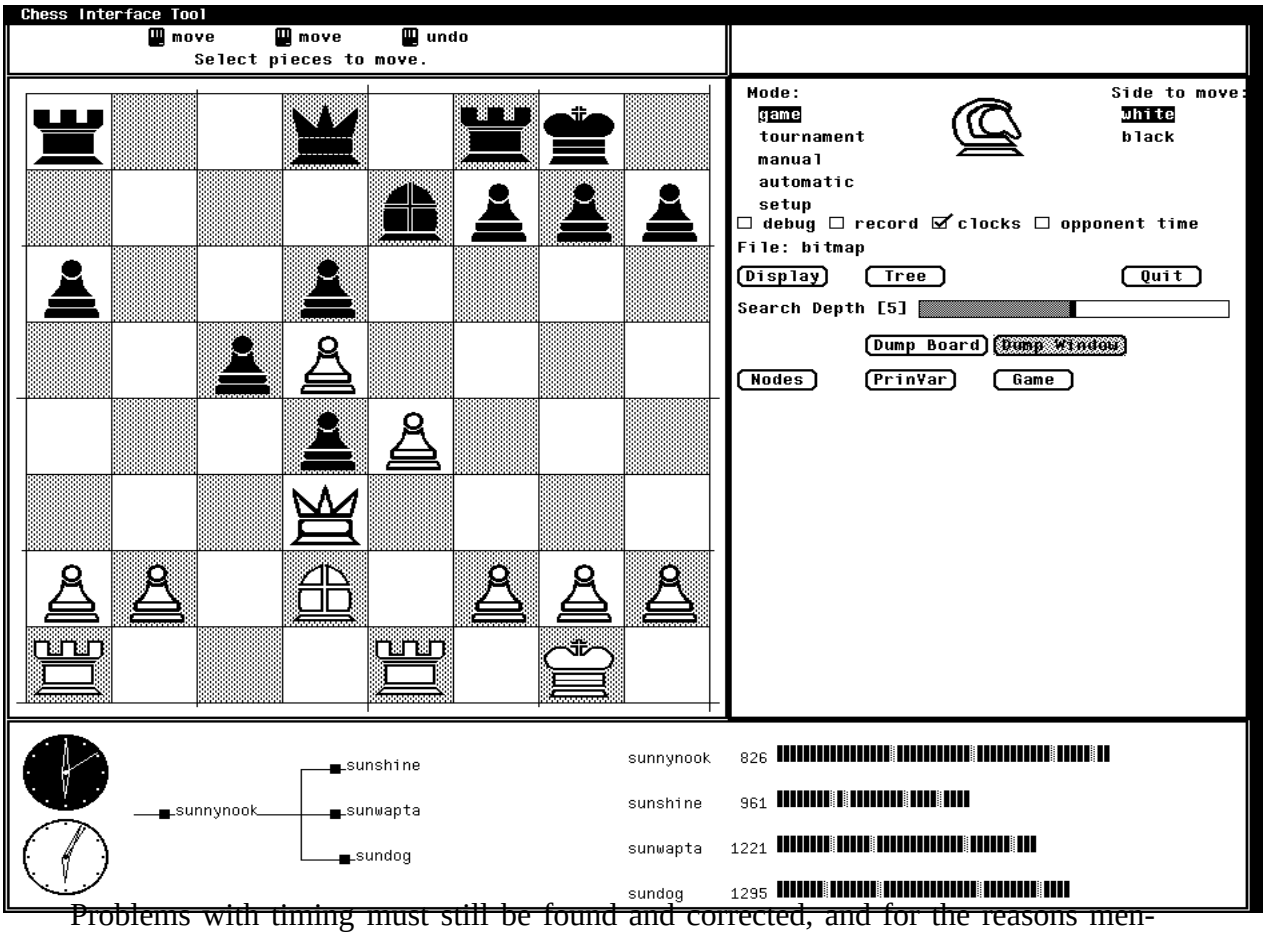
execution characteristics of the program are no longer solely decided by its inputs, but are influenced unpredictably by interactions between autonomous processors, the physical characteristics of the communication medium and by the behavior of other programs sharing these resources. Bugs manifest themselves 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.

To attack the problems of debugging in a distributed environment several mechanisms have been proposed and implemented[3, 4, 22-24]. Garcia-Molina *et al.* argue that distributed debugging is different from debugging sequential programs and then show a bottom-up debugging method that records traces of program execution, examines the traces, and uses them to reconstruct the problem in a simulated environment[3]. LeBlank and Mellor-Crummey save a trace of the relative order of significant events as they occur and use these traces to replay the program[24]. Multibug[23] used a general-purpose computer with UNIX to monitor the node computers, which had a skeletal software environment, similar to our DPM. Multibug worked at a low level, tracing variables and events, but not addressing the larger issues of porting sequential code to their distributed system. The Jade projects monitoring system — Mona[4] — provides tools and graphical displays to monitor at the interprocess communication (Jipc) level. They found that using trace records to recreate erroneous execution was not practical when the system executed for a long time before the error occurred. They suggested that further work should be done in specifying higher level behavior to the monitors. Figure 7 illustrates a high level monitor built for one of our applications using the NMP routines. Our approach to debugging includes these high level monitors as well as a method for

assisting in the porting of sequential code to a distributed environment.

5.1. Graphical Aids

A typical development cycle of an application in the NMP 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 under a single 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 often does not involve recompilation or relinking of the code, but simply a change to the configuration file describing the mapping of the NMP. Recompilation is only required when the processor type changes.



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 program a separate, dedicated machine, to the task of monitoring all processes, and use it to condense and abstract information for human consumption from the other processors in the system. This is done by a ‘monitor-server’ residing on a machine 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 an interface, designed to monitor the execution of a multiprocessor chess program, is shown in Figure 7. In the lower left corner, next to the clocks,

the current NMP node configuration is displayed. The horizontal bars to the right show work completed by the named nodes, with the lighter grey blocks representing synchronization points between iterations of the iterative deepening search. The numbers beside the bars measure how many nodes were searched by each processor during the previous iteration.[2] All processors are now in the midst of their 5th iteration, but the display shows that *sunnynook* may now be doing less work than the others. The rest of the display is then used by the application itself (here Parabelle[25]). 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.

5.2. Robustness

One approach to parallel program development is to avoid the potential for bugs in the code. A technique that has proven useful, is to design timing discrepancy tolerance into the algorithms. Consider, as an example, 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 on receiving the unexpected message. Imagine that a master control process (a parent) does not notice that a ‘work completed’ message has arrived from a child, and so sends it more information that may speed its progress. The child, meanwhile, is waiting for more work, so if the message is tagged it may simply be discarded as opposed to being interpreted as new work.

---

[2] The horizontal bars are a coarse measure of the work completed in terms of trees traversed, the numbers beside the bars count the nodes visited, and is a finer measure of work done.

Polling is another method that should be considered, especially as an alternative to interrupt-driven code. In the NMP environment polling is used to eliminate the danger of blocking because of a lost interrupt. When an interrupt occurs the node must identify the neighbor(s) that have sent messages, by polling all communication paths and reading outstanding messages. This eliminates the danger of deadlock should interrupts be lost when two or more messages arrive simultaneously. In some applications, polling can replace interrupts and can be made less expensive, since no state-change or context-switch is 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.

6. Sample Applications

The facilities described here have been used primarily in experiments with parallel tree-searching algorithms. One vehicle for these experiments were the chess programs Parabelle and ParaPhoenix. Other applications included parallelization of methods to solve the vertex cover problem, a dynamics of articulated bodies study[26], and studies of the overheads in the system[27]. A recent experiment[28] used the NMP to simulate performance of an image compression algorithm. The predicted performance was very close to that obtained on a Connection Machine. More recently a user information management system for configuring multicomputer systems has been built[29].

6.1. Parallel Chess Machines

Parabelle, whose 1983 results are shown in Figure 8, was used to explore the effect of sharing information, to reduce the search times. With distributed machines it is common for one processor to have far more memory than the others, and so is used to hold shared information. However, considerable processing time may be lost when several

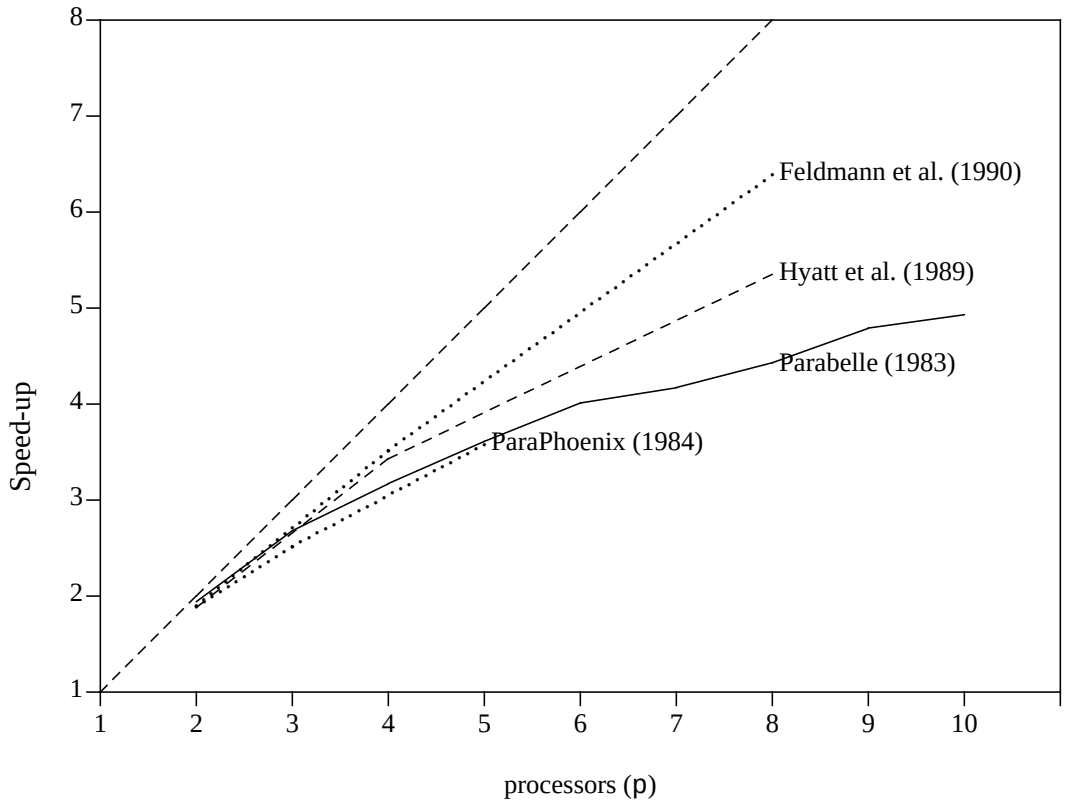


Figure 8. Comparative speedups for selected multiprocessors

processors must await access to the shared global tables. Conversely, the smaller local tables may become overloaded during a search, and so lose their effectiveness. The forerunner of our present NMP system was used to explore the tradeoffs in local/global memory usage[25]. Parabelle itself used a processor tree of depth 1, and the game-trees were searched in a special way by p processors, using the PVSplit algorithm[10]. One of these processors was called the master and had extra duties, such as allocating work to itself and others, and polling them at convenient intervals for their results. For the data here, the processes ran on the DPM, except for the root process, which always uses a general purpose UNIX-based machine. The degrading performance shown in Figure 8 stems from the static nature of PVSplit, which forces the processors to synchronize their activi-

ties as they back out of the tree. The difficulty is not fundamental to the NMP system, as the recent experience with ParaPhoenix shows [30].

ParaPhoenix was the first major NMP application (referred to as VTM in earlier literature[31]). When installed in 1984, it used the same tree-splitting algorithm and processor tree architecture as Parabelle, but a separate process was named the master. Since the master could be put on a separate dedicated machine, it had ample time to measure the activity and effective CPU speed of the others. Thus ParaPhoenix accounted accurately for the synchronization losses in PVSplit, and was used to identify the serious nature of that overhead[31]. Even though, as Figure 8 shows, ParaPhoenix (1984) initially exhibited only comparable speedup to Parabelle, it was a far superior program. Not only did it search 5 times as fast, but also the better heuristics and greater search depth allowed it to achieve better results on standard test suites. This shows that by itself speedup is not an all-purpose measure of performance. By 1989 Schaeffer[30] reported on a dynamic tree-splitting method under NMP in which idle machines were dynamically re-assigned to busy processors. The performance curve for ParaPhoenix thus matched that obtained independently on a Sequent computer by Hyatt et al. (1989) [32], also plotted in Figure 8. Finally, the seemingly best result with that same Bratko-Kopec test suite was obtained by Feldmann et al. (1990) [33], using a Transputer-based system. However, although the speedup curve is impressive, and the testing results were comparable, there remain questions about the validity of limiting the search width to 25 successors (far below the known average search width of 34 moves[25]). This width pruning heuristic is known to be effective but dangerous. It is not clear whether reduced search width helps or hurts speedup, but again it places the measure in doubt. The main problem here is that there is no incentive for anyone to compare their system against the most efficient

uniprocessor version. These excellent speedup results are nevertheless encouraging, since they show the benefit of shared global memory tables. Felten and Otto[34] have also recorded good results including a 101 times speedup on a 256-processor hypercube. Although their basis for comparison is not the same, the benefits of distributed global memory tables come through clearly.

6.2. Additional Case Studies

Other applications include a parallel implementation of the branch-and-bound algorithms for the travelling salesman and vertex cover problems, which were used to investigate the tradeoff between communication, search, and synchronization overheads[27]. It was found that the communication overhead was negligible, even though the workload and communication of optimal results is required, but a new overhead associated with waiting times within the UNIX operating system was uncovered.

Our experiments attempt to measure experimentally some of the costs and overheads involved in distributed processing[27]. 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, including losses within the operating system software itself.

Finally, the NMP facilities have also been used for teaching purposes, specifically in parallel processing and operating systems courses. They are especially useful for learning about parallel computing, because the communications are easily reconfigured and the students do not have to schedule time on a single-user multiprocessor.

7. 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. Although it is 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.

Our experience with the facilities described here show that it is possible to develop and test distributed algorithms under near normal conditions. As long as the results are interpreted correctly, network multiprocessor architectures can provide valuable insight into the behavior of non-existing, new, or unavailable real machines[8]. Algorithms for execution on these architectures can be developed, tested and debugged using this facility. Although the primary purpose of the NMP architecture is to apply several processors to a single application, it can also be used to model large multiprocessor systems and study their processor synchronization and communication delay properties[28].

Future plans for expanding this facility include providing more NMP interconnection paradigms, such as 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.

ACKNOWLEDGMENTS

Marius Olafsson developed the original VTM software in 1984, and helped design the extensions upon which NMP is based. He also played a major role in producing the original report, TR85-9. Financial support from the Natural Sciences and Engineering Research Council of Canada with equipment grant E5722 and operating grant A7902 was important to the success of this research project. The University of Alberta makes the software, and a few sample programs, available via *anonymous ftp* on menaik.cs.ualberta.ca [129.128.4.241] in `pub/nmp.tar.Z`.

References

1. C. Rieger, R. Trigg and B. Bane, 'ZMOB: A New Computing Engine for AI', in *Procs. Int. Joint. Conf. on Artificial Intelligence, Vol 2*, Vancouver, Aug. 1981, pp. 955-960.
2. R. Agrawal and H.V. Jagadish, 'Partitioning Techniques for Large-Grained Parallelism', *IEEE Transactions on Computers*, **C-37**, 1627-1634 (December 1988).
3. H. Garcia-Molina, F. Germano, Jr. and W. H. Kohler, 'Debugging a Distributed Computing System', *IEEE Transactions on Software Engineering*, **SE-10**, 210-219 (March 1984).
4. J. Joyce, G. Lomow, K. Slind and B. Unger, 'Monitoring Distributed Systems', *ACM Transactions on Computer Systems*, **5**, 121-150 (May 1987).
5. F. W. Burton and M. Huntbach, 'Virtual Tree Machines', *IEEE Transactions on Computers*, **C-33**, 278-280 (1984).
6. S.A. Browning, 'A Tree Machine', *Lambda*, **6**, 31-36 (1980).
7. C.L. Seitz, 'The Cosmic Cube', *Communications of the ACM*, **28**, 22-33 (Jan. 1985).
8. 'Myrias 4000 System Description', *Myrias Research Corporation*, Edmonton, May 1984.
9. C. Lam, B.C. Desai, J.W. Atwood, S. Cabilio, P. Grogono and J. Opatrny, 'A Multiprocessor Project for Combinatorial Computing', in *Procs. CIPS Session 82*, Saskatoon, May 1982, pp. 325-329.

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

11. 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, Mar. 1983.

12. D. Ritchie and K. Thompson, ‘The UNIX Timesharing System’, *Communications of the ACM*, **17**, 365-375 (1974).

13. S.J. Leffler, M.K. McKusick, M.J. Karels and J.S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison Wesley, Reading, MA, 1989.

14. B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice Hall, Englewood Cliffs, NJ, 1978.

15. S. Sechrest, ‘An Introductory 4.3BSD Interprocess Communication Tutorial’, *Computer Science Research Group*, Univ. of California, Berkeley, April 1986.

16. T.A. Marsland and S.F. Sutphen, ‘A Heterogeneous Dual Processor’, *Software Practice and Experience*, **10**, 21-28 (1980).

17. T. Breitkreutz, S. Sutphen and T.A. Marsland, ‘Developing NMP Applications’, *Tech. Rep. 89-11*, Univ. of Alberta, Computing Science, March 1989.

18. M. Olafsson and T.A. Marsland, ‘A UNIX Based Virtual Tree Machine’, in *Procs. of the 1985 CIPS/ACI Congress*, Montreal, June 1985, pp. 176-181.

19. G. Fox, ‘Parallelism Comes of Age: Supercomputer Level Parallel Computations at Caltech’, *Concurrency: Practice and Experience*, **1**, 63-104 (September 1989).

20. D.E. Comer, *Internetworking with TCP/IP: principles, protocols, and architecture*, Prentice Hall, Englewood Cliffs, NJ, Second Edition 1990.

21. V.S. Sunderam, ‘PVM: A Framework for Parallel Distributed Computing’, *Concurrency: Practice and Experience*, **2**, 315-339 (December 1990).

22. J.H. Griffin, H.J. Wasserman and L.P. McGavran, ‘A Debugger for Parallel Processes’, *Software Practice and Experience*, **18**, 1179-1190 (December 1988).

23. P. Corsini and C.A. Prete, ‘Multibug: Interactive Debugging in Distributed Systems’, *IEEE Micro*, **6**, 26-33 (June 1986).

24. T.J. LeBlank and J.M. Mellor-Crummey, ‘Debugging Parallel Programs with Instant Replay’, *IEEE Transactions on Computers*, **C-36**, 471-482 (April 1987).

25. T.A. Marsland and F. Popowich, ‘Parallel Game-tree Search’, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **PAMI-7**, 442-452 (July 1985).

26. W. Armstrong, T.A. Marsland, M. Olafsson and J. Schaeffer, ‘Solving Equations of Motion on a Virtual Tree Machine’, *SIAM J. of Sci. and Stat. Comp.*, **8**, s59-s72 (1987).

27. E. Altmann, T.A. Marsland and T. Breitkreutz, ‘Accounting for Parallel Tree-search Overheads’, in *Procs. Int. Conf. on Parallel Processing*, Philadelphia, Aug. 1988, pp. 198-201.

28. X. Li and Y. Chang, ‘Simulating Parallel Architectures in a Distributed Environment’, *Journal of Parallel and Distributed Computing*, **9**, 218-223 (June 1990).

29. A. Singh, J. Schaeffer and M. Green, ‘A Template-Based Approach to the Generation of Distributed Applications Using a Network of Workstations’, *IEEE Trans. on Parallel and Distributed Computing*, **2**, 52-67 (January 1991).

30. J. Schaeffer, ‘Distributed Game-Tree Search’, *Journal of Parallel and Distributed Computing*, **6**, 90-114 (1989).

31. T.A. Marsland, M. Olafsson and J. Schaeffer, ‘Multiprocessor Tree-Search Experiments’, in D. Beal (ed.), *Advances in Computer Chess 4*, Pergamon Press, Oxford, 1985.

32. R.M. Hyatt, B.W. Suter and H.L. Nelson, ‘A Parallel Alpha/Beta Tree Searching Algorithm’, *Parallel Computing*, **10**, 299-308 (1989).

33. R. Feldmann, B. Monien, P. Mysliwietz and O. Vornberger, ‘Distributed Game Tree Search’, in V. Kumar, P.S. Gopalakrishnan and L. Kanal (ed.), *Parallel Algorithms for Machine Intelligence and Vision*, Springer-Verlag, New York, 1990.

34. E.W. Felten and S.W. Otto, ‘A Highly Parallel Chess Program’, in *Procs. Int. Conf. on 5th Generation Computer Systems*, (Tokyo: ICOT), Nov. 1988, pp. 1001-1009.