NMP—A Network Multi-Processor

T.A. Marsland
T. Breitkreutz
S. Sutphen

Computing Science Department
University of Alberta
Edmonton T6G 2H1

Technical Report
TR-88-22
December 1988

ABSTRACT

The report describes an environment for performing experiments in distributed processing. It replaces an earlier (1985) report, reflecting changes in the system and terminology. Our system offers researchers an easy way to design, implement, and test parallel algorithms. It provides software tools which make possible a variety of connection structures between processes. These process structures are said to form a "Network Multiprocessor" (implemented on a local area network of VAX 11/780's, Sun workstations, dedicated MC68000 processor boards, and a MIPS M/1000). We show how these tools have been used both to aid parallel algorithm development and to explore different computer interconnection methods.

December 16, 1988

NMP—A Network Multi-Processor

T.A. Marsland
T. Breitkreutz
S. Sutphen

Computing Science Department
University of Alberta
Edmonton T6G 2H1

Technical Report
TR-88-22
December 1988

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 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 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. On such systems 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. 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 [BuH84]. Tree-structured and hypercube topologies have been proposed to reduce the connections between processors in distributed systems [Bro80]. The advantage of a tree structure 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 [MRC84]. Another advantage of this architecture is that many problems map naturally into a tree or cube structure. These include NP-complete problems such as various combinatorial methods requiring exhaustive search [LDA82] and tree-searching algorithms [MaC82] [Lin83].

2. The Network Multiprocessor

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 Network Multiprocessor (NMP). It is installed on a network of VAX-11/780's, Sun workstations, and a MIPS M/1000 machine. They each run a 4.3BSD based version of the UNIX† operating system [RiT74], see Figure 1. In addition, the system includes ten dedicated Motorola 68000's, without 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). The only portion of the UNIX kernel that was ported to these processors was the networking support. The dedicated machines possess no process management and therefore serve only one application process per physical processor. However, since most of the UNIX run-time library is available, identical code can be run both on the DPM and the UNIX based machines.

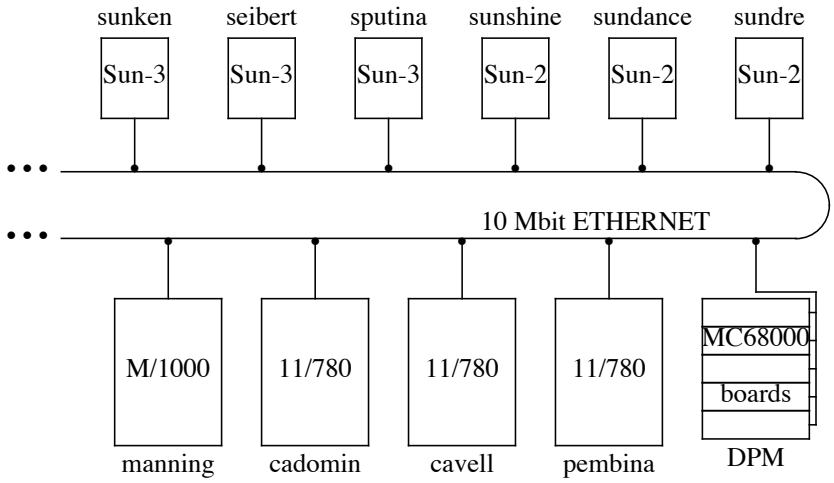


Figure 1. Computing Facilities

The experimenter views the system as a collection of processing elements—each with ample memory, disks and other I/O devices, and arbitrary communication paths with other processing elements. In reality, an NMP is a collection of procedures callable from

† Registered trademark of AT&T in the USA and other countries.

ordinary user programs and a collection of node-servers, one on each physical machine. These node-servers receive requests to create nodes of the NMP according to the description provided by the user. During development the whole NMP might reside on one physical processor before being distributed over the selected physical machines for production use.

The interface to the NMP is a collection of user procedures, written in C [KeR78] 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 NMP being used.

2.1. Standard NMP Routines

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

```
neighbours = NodeInit(Type, ConfigFile)
```

`NodeInit` initializes the node by reading a `ConfigFile` and creating other nodes (processes) as specified in that file. There are two types of node initializations, the first `Type=0` is used in the node that interacts with the user. All other nodes are of `Type=1` and receive the interconnection information from their creator. The configuration file consists of two parts. The first part is an ordered set of descriptor lines, one per node, naming the host process, execution file name and so on, the second part is a connection matrix, showing how the ordered nodes are connected. A node descriptor has the form

```
host_name; bits; executable_file_name; sin; sout; serr;
```

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

`executable_file_name` is the name of the binary for the node;

`bits` is an integer constant whose bits mean the following:

```
Bit #
0    if set, 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 DPM
```

The three last fields contain the names of files to be opened as the node's standard input, standard output and standard error (full path name, or relative to the directory containing the executable file).

The second part of the file is the connection specification. It is a triangular matrix of 0's and 1's, where 1 represents a connection. The full rectangular matrix can be given for clarity, but the top right half will be ignored, since the connections must be bi-directional. For discussion of standard configuration file layout and an example, see

Sections 2.2 and 3. `NodeInit` creates the remaining nodes from the interconnection description received from it's 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 created, control returns to the user's application.

At each node, `neighbours` specifies the number of neighbouring (connected) nodes. Each node in the NMP system is assigned an integer `NodeId`, which is actually its position in the configuration file. The following functions provide a node with its `NodeId` and information about the interconnections.

```
MyId  = GetMyId()
nodes = GetNodes()
conn  = IsConnected(NodeId)
```

`GetNodes` returns the number of nodes configured in the current NMP, and `IsConnected` returns non-zero if the node given in `NodeId` is connected to the calling node.

Nodes may communicate with other connected nodes via the following functions:

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

`NodeId` is the node id of the source or destination. The `Message` is the address of a buffer containing the bytes to be sent, or a space into which to put the received message. These functions return the number of bytes sent or received. Note that normally `RecvNode` waits until all `length` bytes have arrived.

Facilities also exist for managing 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:

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

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

Although interrupt driven message management can work well, there are always cases 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 one process to poll another for pending messages. For this reason, two routines are provided to check the status of messages from neighboring nodes:

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

The nodes that have outstanding messages are returned in the array of node id's, `Nodes`. Normally, these routines return 0 if no message is waiting to be received. Setting `Block` to 1 instructs the routine to wait for the next message to arrive.

There are cases where one would like to change the size of the NMP after the application has started. This could be useful for load-balancing, e.g. deleting a node on a heavily loaded machine and recreating it on a machine with a lighter load. Also, if a node fails for some reason, a new node can be created in it's place thus increasing the fault-tolerance of the whole system. The configuration of the NMP may be changed dynamically by using the following:

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

where `Host` is the name of the physical machine on which the new child is to reside, and `FileName` is the new node's executable file. If file names are given with `Sin`, `Sout`, and `Serr`, they will be opened as the new node's standard I/O streams. `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 standard NMP routines are contained in *NMP Applications Development* [Bre89].

2.2. NMP Superstructures

To eliminate the tedium and risk of making mistakes in constructing standard configuration files for applications where a more regular interconnection structure is used, the NMP Tree and Cube routines are provided. Instead of `NodeInit`, the user may call `TreeInit` or `CubeInit`, which read simplified configuration files, generate the connection matrix, and transparently build a standard configuration file. They in turn invoke `NodeInit`. Each of the initialization functions has corresponding communication, polling, interrupt, and utility primitives. Thus, although users can use the standard routines directly to create virtual multi-processors with arbitrary interconnections, for regular structures like those for Tree and Cube machines, a simplified form of the configuration file can be used with the `TreeInit` and `CubeInit` primitives. Note that the standard NMP primitives allow for other special case interconnection topologies to be implemented if desired.

In early versions of the system, the NMP was called the *Virtual Tree Machine* (VTM) [OlM85a] [OlM85b], because the first implementation was limited to tree structures. The original primitives have been re-written, but for compatibility the NMP Tree configuration file (identical to the old *VTM* configuration file) consists of processor description lines only, as follows:

```
host_name; #children; bits; file_name; sin; sout; serr
```

The `#children` field specifies how many child processes are attached to the node. These children are identified by the implicit "in order" nature of the entries in the `ConfigFile`, as illustrated by Figure 4a. Note that the NMP Tree node ids are relative: the parent is always 0 and children are numbered from 1.

Similarly, `CubeInit` simulates a hypercube processor environment [Sei85] and manages without a connection matrix. Instead, the node descriptor lines are preceded by a number that specifies the dimension of the cube desired. The Applications Development Report gives more details on the NMP Cube and Tree superstructures, and examples of their use [Bre89].

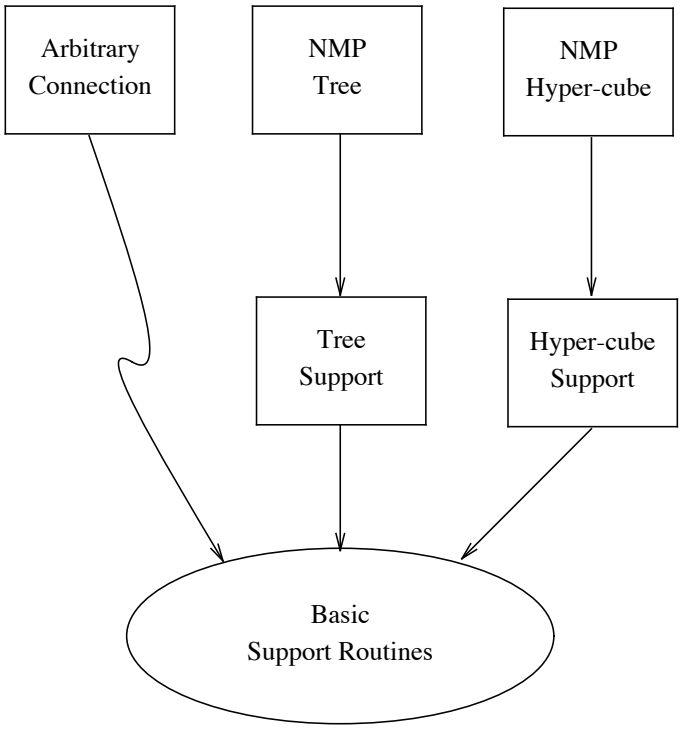


Figure 2. Hierarchy of Network Multiprocessor Implementation

2.3. Implementation of Network Multiprocessors

The NMP environment is implemented using a set of standard NMP routines that allow the creation of virtual multiprocessors with arbitrary interconnection structures, see Figure 2. These standard routines are in turn built onto the UNIX networking primitives. Those primitives allow processes to communicate with a variety of protocols and connection strategies [Sec86]. The current implementation of the support routines uses reliable two-way communication channels (called stream-sockets). Stream-sockets are similar to

UNIX pipes [RiT74], 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; communication between processes is indistinguishable from communication between processors. That is, the 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.

3. Mapping of a Processor Tree

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

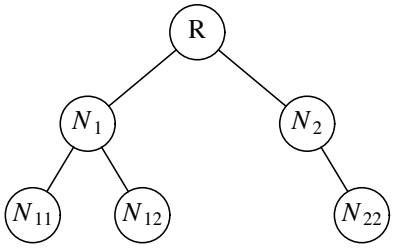


Figure 3. Process Tree

First this configuration must be mapped onto the hardware. 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 pembina
N_{11} on sputina
N_{12} on seibert
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 Suns. The mapping between the virtual machine and the physical hardware is described in the configuration file. To produce a tree machine one can use a simplified form of the standard configuration file, one that omits the connection matrix, as Figure 4a shows. Each line in the tree machine configuration represents a 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. In this example it is assumed that 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 executable filenames would have to reflect the type of binary required for each processor in the configuration. The last three fields of Figure 4a contain the names of files to be opened as the node's standard input, standard output, and standard error streams.

The following configuration file is used to map the virtual processor tree in Figure 3 to specific hardware. The indentation is not required, but helps readability by grouping children with their parents.

```
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 4a. Sample NMP Tree Configuration File

When the root process is invoked by the user on *sunshine*, it calls `TreeInit` to read the above configuration file and translate it to the standard configuration file format as follows:

```
sunshine  ; 0
cavell    ; 0; branch -p1;; out1 ; err1;
seibert   ; 0; leaf -p11;; out11; err11;
sputina   ; 0; leaf -p12;; out12; err12;
pembina   ; 0; branch -p2;; out2 ; err2;
sundre    ; 0; leaf -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 4b. Standard Configuration File

The standard configuration 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*. The server executes the file *branch* (the third field in the configuration entry for the process on *cavell*), gives it the execution parameters specified (here `-p1`), and returns to listen for additional

service requests. It is important to remember that `branch` represents the full path name of the executable file, relative to the user's home directory. 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 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 *pembina* to start up the right branch. Finally, `TreeInit` returns and the application is ready to start work, since all communication paths have now been established.

Note that for some applications it is easier to use the same executable files for all nodes in the NMP, although in this example they have been given different names for clarity. The status of the node may be given by the user when the root is invoked, or may inserted as parameters for children nodes in the configuration file. Examples showing how to develop applications are given in a separate report [Bre89].

The NMP created is shown in Figure 5. With NMP and a typical distributed computing system of Unix based computers on a local area network, several different experiments can be performed at the same time. The Ethernet serves as a shared communication path and processes from different applications may share the same processor.

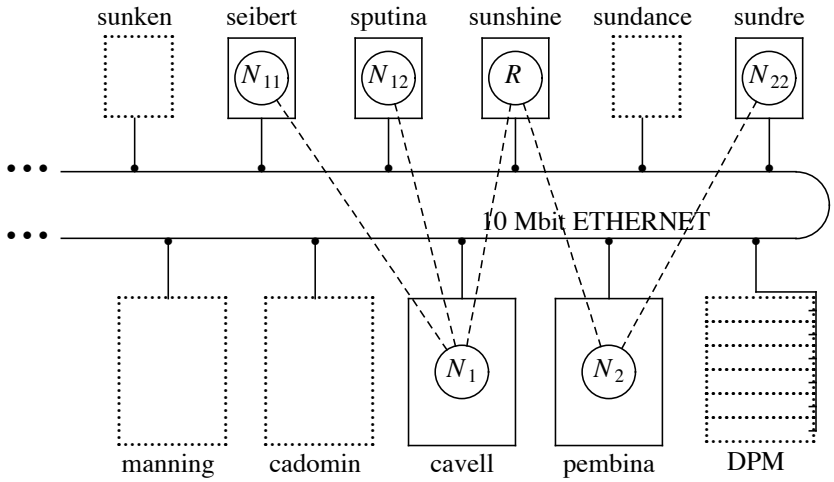


Figure 5. Mapping of Processor Tree onto Selected Hardware Configuration

To illustrate the communication and connection establishment features provided in the NMP environment, the following skeletal C-code segment from an arbitrary interior node is presented in Figure 6. 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 tree branch. Once received, `TreeInit` transmits requests to

start this node's 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`). The parameter `NOINTS` specifies that no interrupts are generated.

The code excerpt of Figure 6 is identical on all nodes in the NMP (except the root where communication with the parent would be replaced with user interaction). Thus, 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.

```
# include <nmp/nmpdefs.h>
int i, n, fanout, length, len;
char *buf;
  .
  .
fanout = TreeInit(0);        /* connection to 'fanout' children */
len = TreeRecvParent(buf, n);            /* receive from parent */

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

Figure 6. Sample communications code.

4. Communication Speed

Table 1 shows the effective communication times using the NMP primitives. These results were obtained by measuring the average transit times for messages of varying lengths. *DPM/DPM* represents two different processor boards (in the DPM) exchanging messages (note that the old 4.2BSD networking was used for these results). *Sun Local* means two processes on the same Sun machine 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 *loopback* up through the software layers to the other process).

VAX/VAX are two independent VAX machines communicating and *Sun/Sun* two independent Sun workstations. Also, *Sun/MIPS* represents a Sun 3 and an M/1000 communication, and *Sun/VAX* means a Sun communicating with a VAX. In addition, these results are averaged over several trials to factor out any peculiar conditions on the network.

As expected, longer messages yield better effective speed, and the effective byte rate increases linearly with message size up to the underlying packet size (1583 bytes). Note the degradation in performance for the 2048 byte messages. These figures include all the software cost of constructing the messages all the way down to packet transmission. Also the cost includes decoding a network packet, reassembling the messages and delivering the message to the application program.

Table 1. NMP Communication Timing

Effective speed in K-Bytes/sec								
	Sun 3/50			VAX 11/780		Mips M/1000		DPM
Length	Local	Sun	VAX	Local	VAX	Local	Sun	DPM
4	1.35	0.76	0.60	0.47	0.33	3.78	1.38	0.8
8	2.80	1.54	1.30	0.97	0.65	8.06	2.89	1.5
16	5.59	3.16	2.43	1.90	1.30	16.06	5.56	2.8
32	10.91	6.16	5.00	3.86	2.46	32.47	10.82	5.7
64	22.25	11.64	9.58	7.51	4.64	61.06	21.05	10.5
128	40.37	20.31	16.26	13.84	8.42	113.27	37.69	16.7
256	70.86	37.40	28.44	25.21	14.36	194.96	62.27	24.9
512	114.36	88.02	53.69	52.98	31.21	285.65	139.43	33.8
1024	270.61	174.40	97.85	95.29	50.44	371.02	221.41	37.8
2048	319.97	173.15	94.33	104.01	58.44	547.84	250.40	47.4

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 low, even when many virtual nodes communicate at the same time. For a typical message size of 32 bytes say, the NMP could accommodate many pairs of communicating nodes. Since each pair only uses 0.2-0.4 percent of the Ethernet bandwidth about 200 such pairs could use the network simultaneously without saturation. However, if needed, higher speed communications could be achieved by simplifying the communication protocols, thus reducing the software overhead and bringing the effective speed closer to the bandwidth of the physical network.

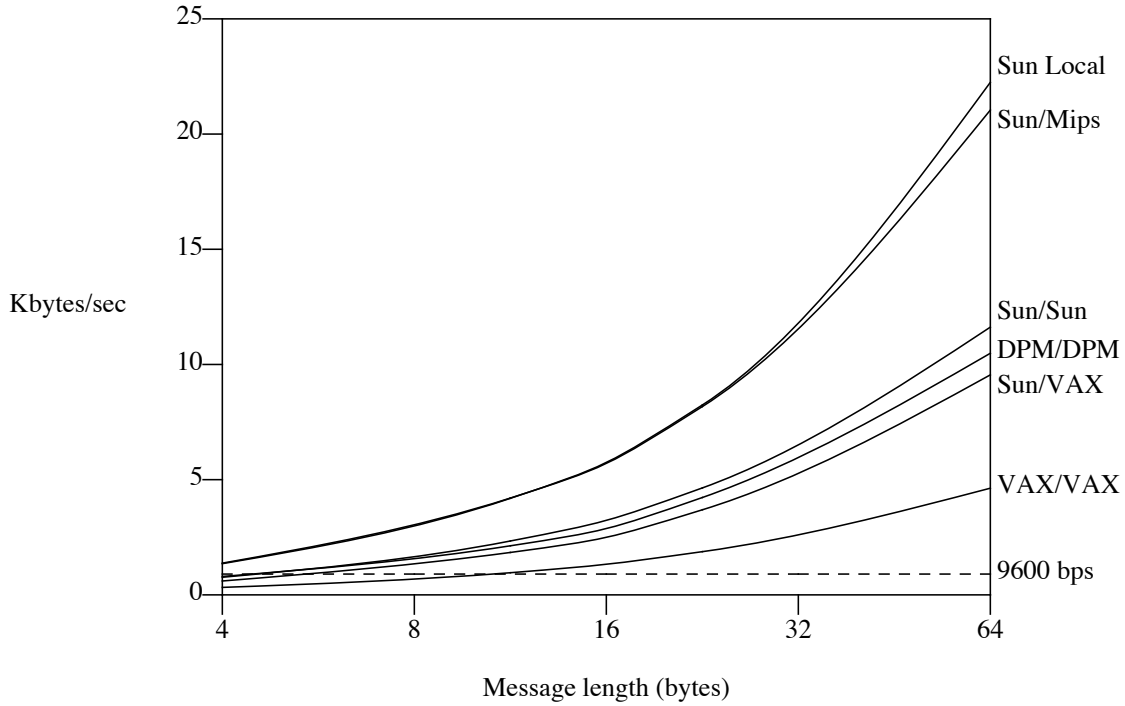


Figure 7. Timing for typical messages

Figure 7 shows a portion of the timing information in more detail. For short messages the fastest communications occur between two processes on the same Sun workstation. Note that DPM/DPM results represent the best from the 1985 report [OlM85a]. The new results illustrate both the benefits of faster processors, MIPS M/1000 and Sun 3/50, over the MC68000 and 68010 processor boards, and the better performance of the 4.3BSD software over the earlier 4.2BSD primitives. Table 1 and Figure 7 show that the dominating cost of the communications is the CPU time spent in the protocol support, 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 VAXen there was never more than about 30% of the CPU time available to each communicating node, because of contention from other users. The timing relationships between the processors changes for longer messages. This is attributed 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 VAXen the message buffers are 1K and on the DPM boards the message buffers are currently only 256 bytes. The dashed horizontal line represents effective speed of 9600 bits/sec and is included for comparison.

5. Debugging

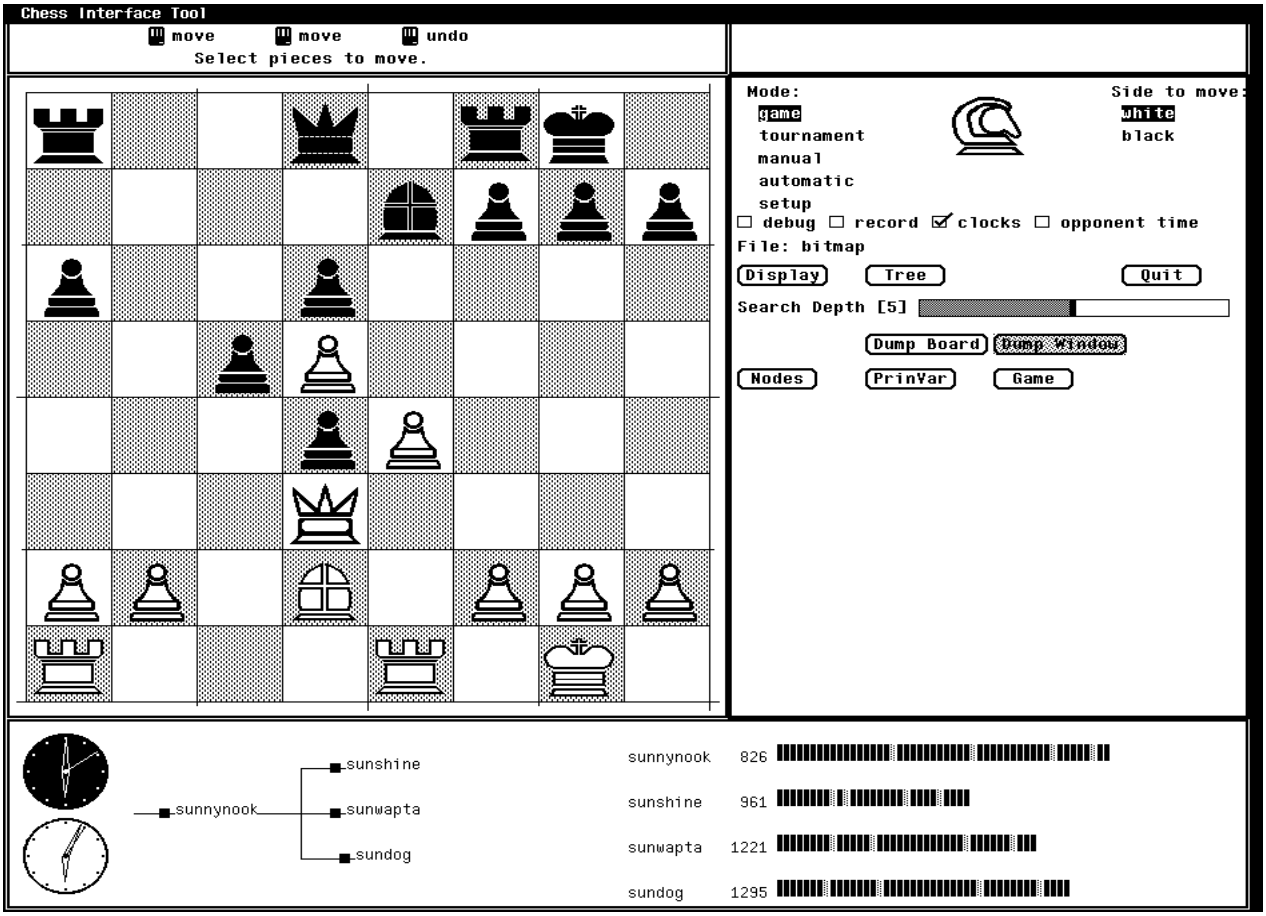
Debugging parallel programs in a distributed environment is more difficult than sequential programs running on a conventional machine. The primary source of this added difficulty is the asynchronous sharing of information in the distributed environment. This sharing (via message passing) between processors 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 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 does not usually involve any recompilation or relinking of the code, but simply a change to the configuration file describing the mapping of the NMP.

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 machine to the task of monitoring all processes. This machine 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 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.

The root node interface may also serve as a monitor, although it may slow down work that may be required of the root. An example of such an interface, designed to monitor the execution of a multiprocessor chess program, is shown in Figure 8. In the lower left corner, next to the clocks, the current configuration is displayed. The horizontal bars to the right show work completed by the named nodes, with grey blocks representing 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. 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 [MaP85]). 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.



5.1. Debugging Aids

One useful technique that has proven useful, is to design timing discrepancy tolerance into the algorithms. An 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 on 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 NMP 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, causing all communication paths to be polled and all outstanding messages to be 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 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. 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 [MaP85] and ParaPhoenix [Sch86] [MOS85] [Sch89]. Also parallelisation of methods to solve the vertex cover problem [BMA87], and studies of the overheads in the system [AMB88], plus a dynamics of articulated bodies study [AMO87].

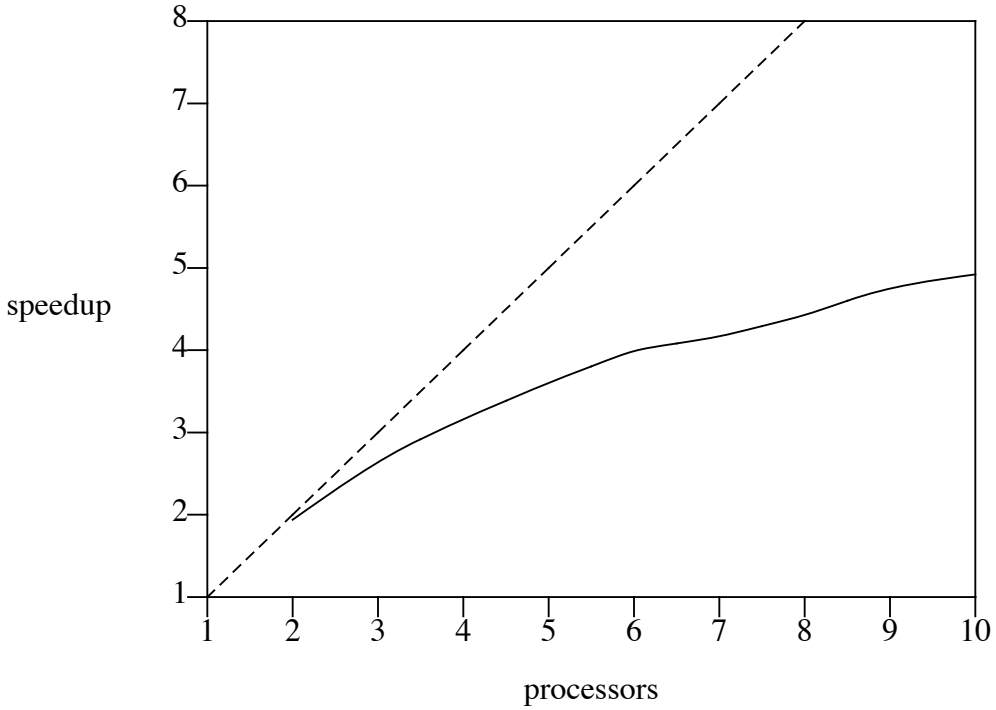


Figure 9. ParaBelle Speedup

6.1. Case Studies

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 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 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 NMP system was used to explore the tradeoffs in local/global memory usage [MaP85]. 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 [MaC82], 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. Later, ParaBelle was implemented on the NMP, and was 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 9. The tree configurations used to produce these results for ParaBelle were run on a six processor DPM, except for the root process which always ran on a UNIX based machine. Similarly, up to three UNIX based workstations were needed to generate the results for the 8 to 10 processor cases.

ParaPhoenix, on the other hand, was the first major NMP application. When installed in 1984, 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 managed 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 in PVsplit, and to identify the serious nature of this overhead [MOS85]. Even so, the master had little to do, so the NMP facility was used to allocate a tree-search process to that machine. That project was followed with experiments involving the dynamic reallocation of processes [Sch86].

Other applications include a parallel implementation of the branch-and-bound algorithm for the traveling salesman problem, which was used to investigate the tradeoff between communication overhead and synchronization overhead. The NMP facilities have also been used for teaching purposes, specifically in parallel processing and operating systems courses.

Our experiments attempt to measure experimentally some of the costs and overheads involved in distributed processing [AMB88]. 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.

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.

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, network multiprocessor architectures can provide valuable insight into the behavior of non-existing, new, or unavailable real machines [MRC84]. Algorithms for execution on these architectures can be developed, tested and debugged using this facility. While the primary purpose of the NMP 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 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.

Acknowledgements

Marius Olafsson developed the VTM software, and prepared 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.

References

[AMB88] E. Altmann, T.A. Marsland and T. Breitkreutz, Accounting for Parallel Tree-search Overheads, *Proc. Int. Conf. on Parallel Processing*, Philadelphia, Aug. 1988, 198-201.

[AMO87] 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(1)*, (1987), s59-s72.

[BMA87] T. Breitkreutz, T.A. Marsland and E. Altmann, Parallel Search of Skewed Trees, TR87-16, Computing Science Dept., Univ. of Alberta, Aug. 1987.

[Bre89] T. Breitkreutz, NMP Applications Development, Technical Report, Univ. of Alberta, Jan. 1989.

[Bro80] S.A. Browning, A Tree Machine, *Lambda 6*, (1980), 31-36.

[BuH84] F. W. Burton and M. Huntbach, Virtual Tree Machines, *IEEE Trans. on Computers C-33(3)*, (1984), 278-280.

[KeR78] B.W. Kernighan and D.M. Ritchie, The C Programming Language, Prentice Hall, Englewood Cliffs, NJ, 1978.

[LDA82] 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.

[Lin83] 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.

[MRC84] Myrias 4000 System Description, Myrias Research Corporation, Edmonton, May 1984.

[MaC82] T.A. Marsland and M. Campbell, Parallel Search of Strongly Ordered Game Trees, *Computing Surveys 14*, (1982), 533-551.

[MaP85] T.A. Marsland and F. Popowich, Parallel Game-tree Search, *IEEE Transactions on PAMI 7(4)*, (July 1985), 442-452.

[MOS85] T.A. Marsland, M. Olafsson and J. Schaeffer, Multiprocessor Tree-Search Experiments, in Advances in Computer Chess 4, D. Beal (ed.), Pergamon Press, Oxford, 1985, 37-51.

[OlM85a] M. Olafsson and T.A. Marsland, Implementation of Virtual Tree Machines, TR85-9, Computing Science Dept. Univ. of Alberta, Edmonton, May 1985.

[OlM85b] M. Olafsson and T.A. Marsland, A UNIX Based Virtual Tree Machine, *Proc. of the 1985 CIPS/ACI Congress*, Montreal, June 1985, 176-181.

[RiT74] D. Ritchie and K. Thompson, The UNIX Timesharing System, *Comm. ACM 17(7)*, (1974), 365-375.

[Sch86] J. Schaeffer, Improved Parallel alpha-beta Searching, *ACM/IEEE Fall Joint Computer Conf.*, 1986, 519-527.

[Sch89] J. Schaeffer, Distributed Game-tree Searching, *J. of Parallel and Distr. Comp.*, to appear, Feb. 1989, 25pp.

[Sec86] Stuart Sechrest, An Introductory 4.3BSD Interprocess Communication Tutorial, Computer Science Research Group, Univ. of California, Berkeley, April 1986.

[Sei85] C.L. Seitz, The Cosmic Cube, *Comm. ACM 28(1)*, (Jan. 1985), 22-33.

Appendix I
NMP Installation Notes

The NMP consists of three libraries, a server, manual pages and documentation, include files, and a set of test routines and demonstration programs. These are contained in the top level directory and various sub-directories. The contents of the top level directory are:

`INSTALL`	- this file, it explains how to install the NMP software.
`copyright`	- information about redistribution rights
`demo/`	- three example programs to illustrate the use of NMP: mandelbrot graphics, the 7-Body problem (sunbody), and a distributed prime number generator.
`doc/`	- documentation on NMP including an application manual and a technical report which gives an overview of the system.
`include/`	- various constants and structures used in the compilations. These are only needed for more sophisticated applications.
`man/`	- the manual pages for the NMP
`src/`	- the C source files for the basic NMP support.
`src.cube/`	- the C source files for the hypercube support.
`src.tree/`	- the C source files for the tree machine support.
`test/`	- a simple test program to test the basic NMP support.
`test.cube/`	- simple test for embedded ring in a hypercube.
`test.tree/`	- a simple test program to test the tree machine support.

Installing the NMP

Since the NMP is intended to be run on heterogeneous distributed processors, the complete installation at a site can be tedious, although it is simple in principle. There are two major components of the NMP—the "*development platform*" consisting of a library and an include file, and the *execution system* consisting of a server. These two elements are independent and need not both exist on any particular processor.

The *development platform* needs to be installed on those machines where applications code will be developed or compiled. A *development platform* must exist for each type of host in your NMP network.

The *execution system* must be installed on each host that will execute NMP applications. The executables for the application code must of course also be installed on each target host.

Development Platform Installation

The steps in this section must be performed for each type of host in your NMP network.

1) **Load the Software.** The first step is to copy the software sources onto the target host. This can be done by whatever method is most convenient (tar(1), rcp(1), or using NFS). To save space one only needs to copy over the `src` (100kB) and `include` (16kB) directories and the `Makefile` (4kB), although having the `test` (36kB) and `demo` (71kB) directories there would also be useful.

2) **Compiling.** Compile the libraries and servers by executing a

```
% make
```

in the top level directory. This will produce a file called `libnmp.a` (48kB) which contains the subroutines that support NMP applications. It will also make the server process (`nmpserver`) which is used in the *execution system*.

3) **Distributing Libraries and Include Files.** The library and include file must be installed so they will be accessible by all NMP developers. Depending on local circumstances you may have to copy the files to other file systems. The `Makefile` is set up install the necessary files on the local host. Before doing a "`make install`" one should check the definitions of `ETC, LIBD,` and `INCD` to see that they are appropriate for your site. You may need special (root) privileges to do this step, as the default paths require write permission in `/usr/include` and `/usr/local/{lib,etc}`.

   If you have to copy the *development platform* binaries to a different machine it is best to be guided by a "`make -n install`" to see what steps are necessary. There are only a few files so hand-to-hand combat is reasonable.

4) **Debugging Support** The makefile includes additional capabilities to produce a profiling version of NMP and a debug version of the library, but these options are not described here.

Execution Systems

The *execution system* consists of a server daemon which is used to start NMP user's application code. This daemon must be installed on each host that will be used for executing NMP programs. In addition, the TCP port that the server and applications use should be registered, and some method of starting the server must be made. Details of how to do these various chores follows.

First one must install the server binary—nmpserver—in an appropriate place. This server should have been compiled along with the libraries in the previous step. The top level `Makefile` will copy it to `/usr/local/etc/nmpserver` by default when one does a "`make install`". Remember that a separate compilation of `nmpserver` will be needed for each different host architecture you have.

The NMP port must be registered on all the machines you wish to have available for NMP nodes. The line:

```
nmp  2000/tcp        # Network Multiprocessor Package
```

goes into `/etc/services` on these machines. We use 2000 to identify the NMP service, any user-defined service number can be used instead. If this number is changed, also change the definition of `NMP_PORT` in `include/nmpdefs.h`, and recompile the package with a 'make' from the base directory. This "wired in" definition is only used when the system call `getservbyname` fails. If you are using the Yellow Pages then only install the service on your YP server machines, and do a `make` in `/etc/yp` to propagate it to the secondary YP servers.

The final step is to start the server and to have it restarted if the machine reboots. The best mechanism for this is to register the server under `inetd`(8). There are minor differences in how to do this depending on what kind of system you are on. If you are using a 4.3BSD system with `inetd` then use

```
nmp stream tcp nowait root /usr/local/etc/nmpserver nmpserver
```

in the file `/etc/inetd.conf`. On the Suns (or other systems based on 4.2BSD), use:

```
nmp  tcp  /usr/local/etc/nmpserver
```

in the file `/etc/servers`. You will have to alter the sample lines to point to the `nmpserver` if you did not install it in the default directory.

Note that the daemon will not be available until the system reboots or the daemon server re-reads its configuration file. (The versions of `inetd` we have cannot expand the daemon table on re-reading the configuration file.)

If you are not using a daemon server, then you will have to start it up in the system run command file, `/etc/rc.local` or `/etc/rc` (Duplicate a daemon startup command already in the file, and substitute the file `/usr/local/etc/nmpserver`). Here, you will have to make the server with `src/OldNmpServer.c` instead of the usual file. See the `src README` and `Makefile`. Alternatively the `OldNmpServer` can be run manually in the background. This will limit the directory searches and execution to the UID of the initiator, but may be useful for testing.

Documentation

The manual pages (in the `./man` directory) should be installed in your local manual section (ours is `/usr/man/manl`). Your local system administrator should be consulted to determine where to put these. The *NMP Application Development* paper and *NMP—A Network Multi-Processor, TR88-22* should be installed with the local documentation (perhaps `/usr/doc/local`).

Test Software

Now you are ready to run the test software in `test`, `test.tree`, and `test.cube`. In each directory, the NMP configuration files will have to be modified to

contain names of your local machines. The configuration files are those that contain a few lines describing which machines and programs to use in the NMP. There are example files called `config` supplied for reference. In addition, there is a "`run`" command that will prompt the user for the topology and will generate a configuration file for that topology and optionally run the test. Remember that the object code image you specify has to be compiled on (or for) the architecture on which it is to be run. See the `README` files in the test directories for more specific instructions. If you encounter any difficulties, please see the *NMP Application Development* paper for more details on compilation and configuration files.

In addition to these test programs there are three demonstration programs supplied in the `./demo` directory. Changing to that directory and doing a `make` will make all the binaries for the current architecture. Appropriate changes will have to be made to the config files before attempting to run the demonstrations. The prime number generation program is an example application that is described in the *NMP Application Development* paper.