Developing NMP Applications

T. Breitkreutz
S. Sutphen
T.A. Marsland

Computing Science Department
University of Alberta
Edmonton, Canada

Technical Report TR89-11
March 1989

Table of Contents

1. Introduction

This paper explains how to write Network Multiprocessor Package (NMP) applications. The technical report *NMP—A Network Multi-processor* [MBS88] contains an overview of NMP, implementation details, more advanced usage information, and instructions for installing the package. NMP replaces its predecessor the *Virtual Tree Machine* (or *VTM*) [OlM85a] [OlM85b]. Section 1 of this paper introduces the Network Multiprocessor, Section 2 describes the basic set of routines, Section 3 describes the NMP Tree Package, and Section 4 the NMP N-cube routines. The Appendix contains a sample NMP (prime number generator) program. The Index of Functions contains the function types, parameters, and the page number where each function is described in the report.

1.1. What Is the Network Multiprocessor Package?

The NMP is a software package intended to make distributed, concurrent applications in a BSD UNIX† socket network easier to write [Sec86]. It sets up a multiprocessor like structure using processes on a UNIX network. The structure of the interprocess communication paths is an arbitrarily interconnected graph. More limited and specific structures may be built on the basic NMP software, as have already been done with the NMP Tree and NMP Cube environments. The Tree routines and Cube routines each provide self-contained sets of primitives built on top of the basic NMP primitives.

1.2. Why use the NMP?

For distributed applications of the client/server model, the Sun Remote Procedure Call package [RPC86] provides a good interface to the socket level of BSD UNIX. RPC, however, is not well suited to multiprocessor simulations or applications of a more concurrent nature. NMP provides procedure calls that allow easy parallelisation of existing sequential algorithms, in which the potential parallelism fits easily into the sequential algorithm.

1.3. Terminology and Notation

In this paper, there are several terms that are used interchangeably for various parts of the NMP system. The acronym NMP can also refer to a single instance of a *Network Multiprocessor*. The computers involved in the NMP are called machines, hosts, or processors. The NMP processes running on the various machines are called nodes or processes, and more specifically may be called children, parents, branches, or the *root* node, which is the single node of the NMP that the user executes and interacts with. Routine names and definitions, example code, example files, and references thereto in the text are given in `constant width font`. Host names are given in *italics*. References to the Unix Programmer's Manual [UPM86] are given in the form *topic(section)*, where *topic* is the manual page name, and *section* is its section of the manual.

2. The Basic NMP Routines

The lowest layer of the NMP is an underlying set of procedures (basic NMP), on which two other layers are built (the Tree and Cube). They are the most general purpose procedures and can be used as a self-contained interprocess communication (IPC) package. The basic procedures allow an arbitrary interconnection matrix between all nodes in the system.

2.1. NMP Nodes

The NMP works by starting up processes on various machines in a local area network. Each NMP node is one process, and is identified by an integer (usually referred to as `NodeId`). The root process is always node 0, and the nodes started later are numbered incrementally from 1.

† UNIX is a trademark of Bell Laboratories

2.2. Configuration Files

Configuration files are the basis for initial structuring of each NMP execution. The interconnection structure and number of nodes in the system can be changed dynamically by the application, but an initial structure must be defined by the start-up configuration file. An example configuration file is shown in Figure 1.

```
#
# 5 processor configuration
#
vax1; 0
vax2; 0; vax/vprog; ; out1; err1;
sun1; 0; sun/sprog; ; out2; err2;
sun2; 0; sun/sprog; ; out3; err3;
mips; 0; mips/mprog; ; out4; err4;
0
1 0
1 0 0
1 1 0 0
0 1 0 1 0
```

Figure 1. Example Configuration File

Configuration files consist of data lines and comment lines (starting with the character #). The data lines are split into two groups: node descriptions and a connection matrix. The first set of data lines contain host and process image descriptions, one line for each NMP node, with the following fields:

```
HostName; Bits; ExecFileName; Sin; Sout; Serr;
```

The `HostName` parameter is the name of the physical processor on which the node should run. `Bits` is a field containing a bit pattern. The right-most two bits are reserved for the NMP (the first for a debug trace, and the second to indicate special hardware), but the others may be used by the application. This field is known internally as the process *info*. The program for each child process to run is given in the `ExecFileName` field. The pathname is relative to the user's home directory on the given processor, or may begin at the root of the file system (`/`). Arguments to the program may be given within the `ExecFileName` field, separated by white space. Finally, the standard input/output channels can be assigned to files or devices. The standard input pathname is given in the `Sin` field, the standard output in `Sout`, and the standard error in `Serr`. The pathnames given for these fields are assumed to be relative to the directory containing the `ExecFileName`, but they may be given with complete path names.

Note that the root node is included in the configuration file, as the first process description line. The only field that is read from this line is the `Bits` field. The host name is filled in with the actual invoking host, but it is customary to include at least the host name field, which is ignored. The arguments and input/output redirection are handled normally from the shell instead of with the configuration file.

Following the node description lines is a connection matrix indicating which nodes are connected to which others. The matrix consists of `1`s (connected) and `0`s (unconnected), separated by spaces. NMP only reads the lower left half of the matrix (since all connections are bi-directional), but the complete matrix may be given if desired. The process interconnection graph for the configuration file in Figure 1 is shown by Figure 2.

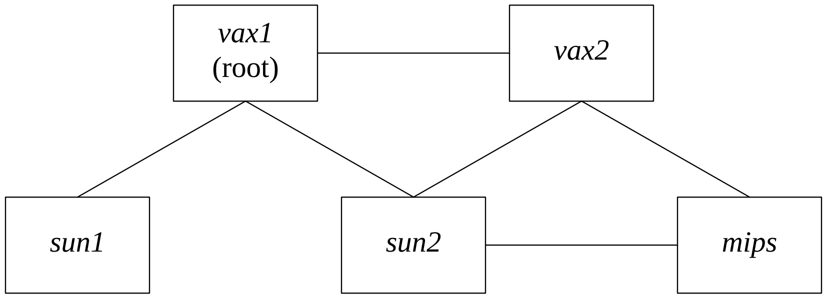

Figure 2. Example Process Configuration

NMP needs to have a path from the root to every node in the configuration, so it will connect the first node in each disjoint part of the configuration to the root node, and provide a warning message to the user. All other connections will be left exactly as indicated in the configuration file.

In NMP applications, nodes are referred to by their *node id* (an integer). The node id is simply the cardinal number of the node description line in the configuration file, starting with the root as node 0. For example, *sun1* in Figure 2 would be identified by the node id of `2`, and *mips* the node id of `4`.

Connection Limits

There is no hard limit on the number of nodes in an NMP configuration, but the interconnection graph is limited by the system-defined limit on file and socket descriptors, usually at least 20 (*getdtablesize(2)* will return the actual number for any particular system).

2.3. Compilation

Most NMP programs simply require that the NMP library be searched in the link edit step. This is accomplished by adding `-lnmp` at the end of your `cc` command, as follows:

```
    % cc NmpProg.c -o NmpProg -lnmp
```

In addition to the regular library a profile library (`-lnmpg`) and a debug library (`-lnmpd`) are available. The debug library reports the progress of the NMP startup, which is helpful for developing new applications, and also provides information for the `dbx` debugger (*dbx(1)*).

Some more complex applications may have to include the header file `/usr/include/nmp.h`. The header file contains the following definitions:

```
        int  NodeType;      /* Internal or Root (user interface) */
        int  SocketMask;    /* Socket mask (for select) */
        int  MyId;          /* My Node Id */
        int *Channel;       /* Virtual Communication lines */
        int *Connect;       /* Connection matrix */
        int  ConnectSize;   /* Size of connection matrix */
        int  ConnectIndex;  /* 2 ** (ConnectIndex) = ConnectSize */

        struct NmpNode *NodeNames;  /* Interconnection structure */
        int NodeNameSize;   /* Size of structure */
        int NodeOffset;     /* Offset from effective to real id */
```

Most of the information given by these globals is available through the NMP utility routines. The `NmpNode` structure is defined as follows:

```
# define M_NAME 80     /* Max node name len */
# define M_HOST 20     /* Max host name length */
# define M_FILE 12     /* Max stream name */
# define M_USER 16     /* Max user name (server) */

struct NmpNode {         /* NMP interconnect information */
    char Host[M_HOST];   /* Host Name */
    char User[M_USER];   /* User login name */
    char Name[M_NAME];   /* Name of node */
    char Sin[M_FILE];    /* Node's stdin */
    char Sout[M_FILE];   /* Node's stdout */
    char Serr[M_FILE];   /* Node's stderr */
    int SecPort;         /* Secondary Port */
    int ModTime;         /* Last mod time */
    int Info;            /* Bits field */
};
```

and contains the information given in the configuration file.

2.4. Errors

Most NMP routines will return a negative number (or a NULL pointer for pointer functions) for an error. An error message will also be printed on the standard error channel.

2.5. Basic NMP Support Routines

The four basic NMP support routines are: NodeInit, SendNode, RecvNode, and NodeClose. These four routines form the core of the system and programs must use (some form) of these four routines.

All basic NMP programs begin concurrent processing by a call from the root node to:

```
int NodeInit(Type, ConfigurationFile)
char    *ConfigurationFile;
```

The root NodeInit causes all the concurrent processes to start. Child nodes also must call NodeInit to synchronise and receive configuration information, but only the root node reads the configuration file. The Type parameter is zero for the root process, and non-zero for child processes. (Often the application source code is the same, and this parameter is itself a parameter to the program or a compilation flag.)

The second parameter, which is only needed for the root node, is the startup configuration file name for the NMP application. Again, this can be a constant, or a parameter to the program. In child nodes, this parameter is ignored because the children inherit their configuration information from the NMP system.

The return value from NodeInit is the number of connections to the current node that were established. If some nodes failed to start up (i.e. their hosts did not respond), then the NMP will continue with the successfuly started subset of the connection graph, and the return values from NodeInit will reflect the actual number of connections.

All messages passed through the NMP are sent using:

```
int SendNode(NodeId, Message, Length)
char    *Message;
```

The data in memory at the location pointed to by Message are sent to the node NodeId (see Section 2.1). The number of bytes to send is given by the Length parameter. SendNode returns the number of bytes successfully sent.

It is important to note that information is always passed as a character array; any other data type pointer may be incompatible across machine architectures (e.g. VAX integers are stored differently in memory from most other machines). A solution to this problem is to use the network byte ordering macros (see *ntohl(3N)*) to store integer and short data for transmission. For example, code using the *host to network short* macro to send an array of short integers might look like:

```
short   data[DATA_SIZE];

for (a = 0; a < DATA_SIZE; a++)
    data[a] = htons(data[a]);
SendNode(dest, data, sizeof(data));
```

If the byte ordering on all the machines is the same, then the same segment could be simplified to:

```
SendNode(dest, data, sizeof(data));
```

Another, more general, mechanism for ensuring data portability is the eXternal Data Representation (XDR) package developed at Sun Microsystems and publicly distributed [XDR86].

To receive information sent from `SendNode`, use

```
int RecvNode(NodeId, Message, Length)
char    *Message;
```

`Length` bytes are received from `NodeId` and stored at the memory location pointed to by `Message`. Returned is the number of bytes received, zero if the node is no longer connected, or negative for other errors. `RecvNode` will only return with a partial message if the sender terminated before completing transmission. Otherwise, `RecvNode` always blocks until the entire message is read. To check for incoming messages without blocking, use the polling routines described below (Section 2.6).

The following code receives the integers as sent in the above `SendNode` example:

```
RecvNode(src, data, sizeof(data));
for (a = 0; a < DATA_SIZE; a++)
    data[a] = ntohs(data[a]);
```

Concurrent processing with the NMP is terminated by

```
NodeClose()
```

Child processes are terminated by this call, but the root node can continue processing and may begin more concurrent segments by calling `NodeInit` again.

Example: Simple Message Passing Application

It is possible to write an NMP application using only the routines listed so far. The following example of an NMP program uses only the four basic routines to create a connected pair of processes and pass a single message. The source code for the root and branch processes is in the same file, distinguished with compiler conditionals.

```
# include <stdio.h>

main() {
     int     neighbours;
     char    buf[16];

     neighbours = NodeInit(TYPE, "config");
# if TYPE == 0
     if (SendNode(1, "Hello, there.", 13) < 13)
          fprintf(stderr, "unable to send");
# else
     if (RecvNode(0, buf, 13) < 13)
          fprintf(stderr, "unable to receive");
     if (strcmp(buf, "Hello, there."))
          fprintf(stderr, "received wrong message");
# endif
     NodeClose();
}
```

Figure 3. Example NMP Program

The corresponding configuration file, `config`, could be:

```
vax1;0
vax2;0;branch;;out1;err1;
0
1 0
```

The file `branch` must reside in the user's home directory on `vax2`. The two program images are compiled with:

```
% cc -DTYPE=0 -o root test.c -lnmp
% cc -DTYPE=1 -o branch test.c -lnmp
```

and invoked by

```
% root
```

The Appendix contains a more complete example that uses more of the NMP routines.

2.6. Polling

If your application needs to check for incoming messages without using the blocking `RecvNode` routine, or if it is waiting for messages from more than one sender, the polling routines may be used. The three polling related routines do not affect the data in the communication paths in any way.

To poll all the neighbouring nodes, use

```
int PollAll(Nodes, Block)
int     Nodes[];
```

If any data are waiting to be read from nodes, the sender node ids are returned in the array `Nodes` (which should be large enough to contain all nodes which may have incoming messages). `PollAll` returns the number of nodes with messages pending to be received. The `Block` parameter tells `PollAll` whether to wait for an incoming message if there are none when it is called. A zero `Block` parameter will cause `PollAll` to check the neighbouring nodes only once, otherwise it waits until a message is pending.

To check an explicit node for pending messages, use

```
int PollNode(NodeId, Block)
```

A return value of zero indicates no messages are waiting to be received, non-zero indicates otherwise. `Block` has the same meaning as for `PollAll`.

The following procedure allows you to look at an incoming message before receiving it.

```
int CheckNode(NodeId, Message, Length)
char    *Message;
```

A copy of the byte stream from `NodeId`'s incoming message queue is placed in `Buffer` without affecting the status of the input queue. It attempts to read in up to `Length` bytes, and returns the number of bytes copied. Unlike `RecvNode`, `CheckNode` does not wait until `Length` bytes arrive, it will copy a smaller size message if it is available.

If you want to be more selective in your node polling, use

```
SetMask(NodeId)
```

and

```
ClearMask(NodeId)
```

These two procedures are used to manipulate the masks used to determine which nodes the polling operations listen to. If a particular node (`NodeId`) has been temporarily eliminated (via `ClearMask`), a polling operation (`PollAll` or `PollNode`) will never indicate that there is any input waiting to be received from that node. `SetMask` adds the given node back to the mask (see *select(2)*).

Detecting a Node Disconnection

If the polling routines detect a disconnected node, they will return as if there is a message waiting to be received from the node. However, no information will be available from `RecvNode` (or `CheckNode`) which will return a length of zero bytes read. There is no way to detect immediately that a node is no longer connected with the polling routines.

2.7. Other NMP Routines

Most NMP applications only need the above routines. See the Appendix for an example of a prime number generator that has been parallelised using the basic routines. The following sections describe primitives provided for more complex applications and improved error handling.

2.7.1. Interrupts

An alternative to polling is to use interrupts. If a node wishes to be interrupted on receipt of a message, it must establish an *interrupt handler* routine. Interrupting messages are received exactly the same as normal messages. The only difference is that the receiver is notified of the message's presence by a call to its interrupt handler. NMP interrupts are actually SIGIO signals (see *sigvec(2)*) generated by sockets with their `FASYNC` flag set (see *fcntl(2)*).

To set up an interrupt handler on a node planning to receive interrupts, use

```
SetHandler(handler)
int     (*handler)();
```

This routine also *enables* interrupts. The argument is the name of your interrupt handling procedure The signal handler routine will be called directly by the system, it is not intercepted by the NMP software

To hold incoming interrupts temporarily (for critical sections of code), use

```
HoldInt()
```

Held interrupts may be lost if they are not the first to arrive.

```
ReleaseInt()
```

releases any held interrupts. Note that the above two routines do not nest.

The following routines completely disable or re-enable the receipt of interrupts.

```
DisableInt()

EnableInt()
```

Interrupts are not queued when disabled; they are thrown away.

Interrupts are automatically be *held* before entering, and *released* on exit. Interrupts may be lost if more than one interrupt is received while the interrupt handler is executing. Generally speaking, a node should not rely on being able to receive more than one interrupt without acknowledgement to its sender.

The `SendNode` and `RecvNode` routines may be considered atomic with respect to interrupts. That is, their underlying system calls are restarted in the event that an interrupt occurs before they are complete. Therefore, it is not necessary to disable interrupts before a call to any of the communication routines, but other system calls may need protection from premature exit caused by incoming interrupts.

The interrupt mechanism used can cause multiple interrupts for a single message if the message is larger than the internal buffers. Because of this programs receiving large messages should accumulate the message as it arrives. An example interrupt handler that does this is shown here:

```
int NewWork = 0;
char buf[LargeSize], *bp = buf;

handler()
{
    register int len, remain = &buf[LargeSize] - bp;

    len = CheckNode(neighbour, bp, remain);
    if (RecvNode(neighbour, bp, len) != len) {
        /* handle error condition */
    }
    bp += len;
    if (remain == len) {
        NewWork++;
        bp = buf;
    }
}
```

```
main()
{
    char workbuffer[LargeSize];
    ...
    /* read in initial work */
    if (RecvNode(neighbour, workbuffer, LargeSize) != LargeSize) {
        /* handle error condition */
    }
    SetHandler(handler);

    while (1) {
        DoSomeWork();
        if (NewWork) {
            /* switch work loads */
            HoldInt();
            bcopy(buf, workbuffer, LargeSize);
            NewWork--;
            ReleaseInt();
        }
    }
}
```

This contrived example works on the old data (in `workbuffer`) while the new work is assembled in `buf`. It also blocks out interrupts in the critical section of shuffling buffers.

2.7.2. NMP Utilities

Although the NMP data structures are not strictly opaque, it is a good idea (and easier) to use the following routines to access the internal state of the NMP structures.

- A node can determine the name of the machine it's running on with `GetHost`.

  ```
  char *GetHost()
  ```

- To find out the host machine name of another NMP node, use

  ```
  char *GetAnyHost(NodeId)
  ```

  `NodeId` is the other node's id number.

- To find out the total number of nodes in the NMP, use

  ```
  int GetNodes()
  ```

- A node may determine its own node id by calling

  ```
  int GetMyId()
  ```

- The `Bits` (*info*) configuration file field is obtained by calling

  ```
  int GetInfo()
  ```

The basic `SendNode` and `RecvNode` routines do not check the connectivity of their destination and source nodes. An attempt to send to or receive from a non-existent or non-connected node will return an error code and produce a message on the standard error channel. The following routines are used to determine the status of the NMP connection configuration. Nodes configured as connected may not be connected because of an error in starting up the NMP or if one has already stopped.

- To find out if a node is connected to the current node, call

  ```
  int IsConnected(NodeId)
  ```

- The configured connection between two other nodes may be determined by

  ```
  int AreConnected(NodeId1, NodeId2)
  ```

  A non-zero return indicates a connection, zero indicates no connection.

The NMP assumes that if some of the machines in the configuration are unavailable you wish to continue with the remaining nodes.

- If you need to know if the entire configuration was successfully started, use:

  ```
  int MadeConn()
  ```

  It returns 1 if the NMP setup was successful, zero if one or more nodes failed to start up.

- The current connection matrix can be printed with

  ```
  PrintConnect(stream)
  FILE    *stream;
  ```

  which will place it on the standard I/O output `stream`.

- If you wish to use your own node numbering scheme, `SetNodeOffset` is provided to cause all references to nodes to have `Offset` added (i.e. when the user refers to node 1, the real id of the node will be 1 + `Offset`). The offset is also subtracted before returning any node ids to the user.

  ```
  SetNodeOffset(Offset)
  ```

2.7.3. NMP Clock

Two clock routines are provided for system-consistent timing of programs. They are

```
clock_start()
```

and

```
clock_stop(rt, ut, st, it)
int    *rt, *ut, *st, *it;
```

These calls may be nested, `clock_stop` terminating the most recent instance of `clock_start`. The real time is returned in `rt`, the user time in `ut`, the system time in `st`, and the idle time `it`. All times are in milliseconds. Real time is the elapsed time between the two system calls. User time and system time are approximations of CPU time in the user's program and in system calls. Idle time is calculated by `clock_stop` as the difference between real time and combined user and system time. See *gettimeofday(2)* and *getrusage(2)* for more details about these units.

2.7.4. Dynamically Reconfiguring Your NMP System

If the application needs to change its configuration while running, it may add nodes and connections. Nodes can delete themselves by calling `NodeClose`. Connections cannot be deleted, but may be ignored by using the select mask manipulation routines (Section 2.6). The following procedure creates a new node to participate in the NMP and establishes a connection between it and the node issuing the call.

```
int AddNode(HostName, ExecFileName, Sin, Sout, Serr, Bits)
char    *HostName;
char    *ExecFileName;
char    *Sin, *Sout, *Serr
```

`HostName` is the name of the physical machine on which the node is to run, `ExecFileName` is the name of the node's executable file, `Sin`, `Sout`, and `Serr` are the node's standard I/O streams, and `Bits` is the user-definable information passed to the created node. The node id of the new node is returned. The interconnection structures are updated appropriately, but only in the calling node and the newly created node. The rest of the nodes are not informed; their calls to `GetNodes` or any other utility routines will not show the existence of the new node. The new node is assigned a node id one higher than the highest id that the current node is aware of. That is, the initial configuration information is distributed to all nodes in the system, but only the nodes involved in any subsequent changes know about the new changes. See Section 2.2 on Configuration Files for more details.

To create a new connection between existing nodes, use

```
int AddConnect(Type, NodeId)
```

There are two possibilities: `Type=0` is the active part of the connection, and `Type=1` is the passive part. The nodes involved must synchronise on the call to `AddConnect` and must know who is active and who is passive. The connection matrix is updated only in the two nodes involved. The other nodes know nothing of this new connection. At the moment, the passive part of the connection does not verify which node it is connecting to, so passive calls to `AddConnect` should be done carefully to avoid misconnection.

Warning

Nodes cannot distinguish `AddConnect` requests and their initial startup message. If a node is waiting for its startup message and receives an `AddConnect` request instead, it will hang, expecting the startup information as well. Therefore, applications must await the completion of the NMP startup before attempting `AddConnects` to nodes deep in the interconnection graph.

3. The NMP Tree

The NMP Tree layer is an extension of the NMP designed specifically to simulate processor trees, allowing a simpler configuration file and a more descriptive set of primitives to exchange messages. They consist of a separate group of routines that are built on, but can be used independently of, the Basic NMP Routines. The Tree routines are included in the regular `-lnmp` library, so compilation is identical with that for basic NMP.

3.1. NMP Tree Nodes

Nodes in an NMP Tree are numbered relative to their parent. That is, nodes are identified by parents with a number from 1 to *n*, where *n* is the number of children the processor has. Child number 0 is the node's parent, if it has one. (The root node has no parent.)

3.2. Tree Configuration Files

The configuration files for NMP trees differ from those used by the basic NMP routines.

```
#
# Tree File
#
sun1; 2; 0
        sun2; 2; 0; Ihelper p1 p2; ; out1; err1;
                sun3; 0; 0; Lhelper; ; out11; err11;
                sun4; 0; 0; Lhelper; ; out12; err12;
        sun5; 0; 0; Lhelper; ; out2; err2;
```

Figure 4. Sample Tree Configuration File

The indentation is not required but can be used to show the structure of the NMP tree. Here, *sun1* is the root node, with two children, *sun2* and *sun5*: *sun2* also has two children, *sun3* and *sun4*. The configuration file format is the same as for regular NMP's except that there is no interconnection matrix, and each line has an extra field for the node's number of children. The fields are:

```
HostName; NChildren; Bits; ExecFileName; Sin; Sout; Serr;
```

Each node from the description lines is assigned to the last node encountered still waiting for children.

3.3. Basic Tree Routines

Instead of `NodeInit`, NMP Trees use

```
int TreeInit(Type, Config)
char    *Config;
```

Again, `Type` is `0` for the root node, and `1` for interior nodes of the tree. `Config`, required only for the root node, is the name of the configuration file (described above). `TreeInit` returns the number of children successfully started.

For interior or leaf nodes, `TreeInit` blocks, waiting for a request from its parent node to start. Once such a request is received, it accepts the tree machine configuration from its parent and starts up its own children (if it has any).

The basic NMP routine `NodeClose` is used to terminate a Tree NMP process.

3.4. Tree Communication

The following procedure sends a message to a child node.

```
int TreeSendChild(Child, Message, Length)
char    *Message;
```

`Length` bytes of data pointed to by `Message` are sent to child number `Child`. `TreeSendChild` returns the number of bytes successfully sent, or a negative return code on error.

To receive a message from a child,

```
int TreeRecvChild(Child, Message, Length)
char    *Message;
```

waits for and receives `Length` bytes from its `Child` and stores the message in `Message`. If the sender has terminated without placing enough data in the stream, the partial message is read and its length returned. Otherwise, success is indicated by a positive return value (actually the full length).

The following two routines are provided for clarity, and operate exactly as the above procedures with a `Child` parameter of zero (`Child` zero is really the parent).

```
    int TreeSendParent(Message, Length)
    char    *Message;

    int TreeRecvParent(Message, Length)
    char    *Message;
```

The root node has no parent and will thus generate an error if it attempts to communicate using these two routines.

3.5. Polling in the Tree

The following routines are used to check the status of the communication paths. None affect the data on the send/receive queues in any way.

To check for any incoming messages, use

```
    int TreePoll(ChildReady, Block)
    int     ChildReady[];
```

If data are queued, waiting to be read from a child node (or the parent), its index is returned in `Chil-dReady`. That is, if a child *i* (or the parent if $i = 0$) has sent a message that has not yet been read, *i* is entered into the array `ChildReady`. This procedure returns the number of "ready" children. If `Block` is true, the process waits until data arrives from at least one child or the parent.

For polling only the parent,

```
    int TreePollParent(Block)
```

returns zero if no message outstanding, non-zero otherwise.

The following procedure checks a single child for pending messages:

```
    int TreePollChild(Child, Block)
```

Zero is returned when no data are pending, non-zero otherwise.

To peek at incoming data without receiving it, use

```
    int TreeLook(Child, Buffer, Length)
    char    *Buffer;
```

Up to `Length` bytes of data waiting to be read from `Child` are copied into `Message`. The number of bytes successfully copied is returned: the same bytes will be obtained on the next look or receive. To examine a parent's pending messages, set the `Child` parameter to zero.

As for the regular NMP, these polling routines will return successfully if a node is disconnected. The next receive or look will return zero bytes, indicating the untimely demise of the sender.

3.6. Other Tree Routines

The following routines supplement those described in Section 2.7 for the basic NMP.

3.6.1. Utilities

The following routines allow access to the internal tree data structures.

- The current node's depth (its distance from the root) in the processor tree is returned by

  ```
  int TreeGetDepth()
  ```

- The *sibling number* of the current node is obtained with

```
int TreeGetNumber()
```

This is the number by which the parent of the current node knows it.

- The number of children *configured* for the current node is returned by

```
int TreeGetSlaves()
```

Note that this number need not be the same as the number of children really available.

- The maximum depth of the tree is reported by

```
int TreeGetMaxD()
```

- The following two routines report the number of nodes in the processor tree.

```
int TreeInodes()
```

returns the number of internal nodes in the subtree rooted at the current node, and

```
int TreeLnodes()
```

returns the number of leaf nodes. For the example tree configuration given in Figure 4, `TreeGet-MaxD` would return 3, `TreeInodes` would return 2, and `TreeLnodes` would return 3.

- To print out a textual representation of the processor tree on the standard output, use

```
TreePrint(Index)
```

If only the subtree rooted at the current node is desired, use `Index=GetMyId()`. If the entire tree is to be printed, use `Index=0`.

3.6.2. Dynamically Reconfiguring the NMP Tree

New nodes can be dynamically added to the NMP Tree configuration by using

```
int TreeAddChild(HostName, ExecFileName, Sin, Sout, Serr, Bits)
char    *HostName;
char    *ExecFileName;
char    *Sin, *Sout, *Serr;
```

Each call connects a new child to the calling node. Only one child may be added at a time. (There is no provision for adding a whole branch.) The caller must specify explicitly the same information as is found in the configuration file for statically configured nodes, i.e. the `HostName` on which the node is to reside, the `ExecFileName` of the NMP program, the names of the files to receive the nodes' standard I/O streams `Sin, Sout,` and `Serr`, and finally the user-defined parameter `Bits`. `TreeAddChild` returns the new number of children if successful, otherwise a negative number.

Note that the process of adding a new node can be time consuming. The node's executable image must be read from disk, and some processes may be swapped out to make room for it on the host machine.

```
int TreeDelChild(ChildNo)
```

disconnects the specified `ChildNo` from the current node. Note that this routine only updates the data structures and closes the socket. It is the caller's responsibility to handle the child's exit gracefully and make sure that no data are in the send/receive queues on either end. Note that the parent must initiate the disconnection. The child simply calls `NodeClose`. The number of remaining children is returned.

4. The NMP Cube

The NMP Cube layer is provided for applications desiring to view the execution environment as a hyper-cube. They provide a slightly different interface as well as a simpler configuration file (like the NMP Tree). Messages may be assigned different types and a separate message queue is maintained for each type. Other differences are that messages are limited in length and the communication channels are designed to support true messages (as opposed to byte streams). Routing within the cube is performed automatically, but synchronous with calls to any of the cube support primitives. Thus some of the communications bandwidth of any node may be taken up for routing messages between other nodes. These routines are designed to provide compatibility with the programmer's interface to the Intel iPSC and the CALTEC Cosmic Cube environments [Sei85].

4.1. NMP Cube Nodes

In the hyper-cube environment, nodes are numbered from 0 to $2^n - 1$ where n is the dimension of the cube. The user, as in the regular NMP, interacts with node 0.

```
#
# Example of a 3-cube
#
3
sun1; 0;
sun2; 0; cube; ; out1; err1;
sun3; 0; cube; ; out2; err2;
sun4; 0; cube; ; out3; err3;
vax1; 0; vcube; ; out4; err4;
sun5; 0; cube; ; out5; err5;
sun6; 0; cube; ; out6; err6;
sun7; 0; cube; ; out7; err7;
```

Figure 5. Example Cube Configuration

4.2. NMP Cube Configuration Files

The configuration for the cube is described by the cube dimension followed by the names of machine/process instances, one per line in the file. The format is identical to the basic configuration file, but with the dimension of the cube at the start and without the connection matrix. Each line in the configuration file represents a node in the cube and contains the same information as the general configuration entries.

4.3. Basic Cube Routines

The NMP Cube initialisation routine

```
int CubeInit(Type, Config)
char    *Config;
```

starts the processes running and establishes the hyper-cube interconnection. `CubeInit` must be called from every process involved in the cube and `Type` distinguishes between a call from the root node (`Type=0`) where `Config` is the name of the configuration file, and the other nodes (`Type=1`) where `Config` is ignored.

To exit an NMP cube node, call

```
CubeClose()
```

4.4. Cube Communications

Internode messages in the cube environment may be assigned different types: thus a node can receive messages in an order different from the order in which the messages were sent. Since routing is done in the user process, all calls to the message primitives result in the interrogation of all message channels and forwarding of messages that are destined for nodes other than the one issuing the call. Notice also that no routing takes place unless a call is made to a message primitive. (See the discussion on `CubeProbe` below.)

```
int CubeSend(Node, Type, Message, Length)
char    *Message;
```

sends `Message` of a specified `Type` to the given `Node`. The size is given as `Length` and is currently limited to 256 bytes. Notice that the message may have to pass through as many as $n-1$ intermediary nodes (where n is the cube's dimension) to reach its destination. `SendNode` returns the number of bytes successfully sent or negative on error.

```
int CubeRecv(Node, Type, Message, Length)
int     *Node;
char    *Message;
```

receives the oldest queued `Message` of a specified `Type`. The originating `Node` is returned in place and the size of the message is returned. `CubeRecv` will block until `Length` bytes are received.

To send a message to all the nodes in the cube, use

```
int CubeRingSend(Direction, Type, Message, Length)
char    *Message;
```

`CubeRingSend` is like `CubeSend` except that the message is always sent to the next/previous node in a fixed ring embedded in the cube. If `Direction` is 0, the message is sent to the *next* node in the ring, and if it is 1 the message is sent to the *previous* node in the ring. The ordering in this context is on numerical node ids.

```
int CubeProbe(Node, Type, Length)
int     *Node;
int     *Length;
```

checks for a message of the specified `Type`. If a message is waiting to be received (either on the queue or on the network) `CubeProbe` returns 1 and returns the sending `Node` and the message `Length` in place. If nothing is pending, zero is returned.

This routine should be called frequently even if no message is expected at this node, since it may be involved in routing messages between other nodes in the cube.

Error Values

The above routines return specific error codes:

HEADER_ERROR	-100	Error receiving message header
BODY_ERROR	-101	Error receiving message body
ROUTE_ERROR	-102	Error in routing message
QUEUE_ERROR	-103	Message Queue Overflow

4.5. Cube Utilities

Here are the utilities specific to the cube environment:

- The dimension of the cube is obtained with

```
int CubeDim()
```

- The number of messages waiting to be read from a node's input queue is returned by

```
int CubeQsize()
```

- For a graceful error exit, call

```
CubeExit(Code, fmt, a1, a2, ....)
char    *fmt;
```

The `Code` specifies whether to exit without trying to clear up messages in transit (`Code=1`) or wait for ten seconds while trying to route messages to their destinations (`Code=0`). A terminating message is printed on the standard error channel as specified in *printf(2)* form.

Acknowledgements

This report was inspired by the VTM manual pages written by Marius Olafsson, who also suggested some of the examples and wrote the VTM package. Numerous students at the University of Alberta made helpful suggestions.

References

[RPC86] Remote Procedure Call Programming Guide, Sun Microsystems, Inc., Mountain View, Feb. 1986.

[UPM86] 4.3BSD UNIX Programmer's Manual, Univ. of California, Berkeley, April 1986.

[XDR86] External Data Representation Protocol Specification, Sun Microsystems, Inc., Mountain View, Feb. 1986.

[JLS87] J. Joyce, G. Lomow, K. Slind and B. Unger, ‘‘Monitoring Distributed Systems,’’ *ACM Trans. on Computer Systems* **5(2)**, 121-150 (May 1987).

[MBS88] T. A. Marsland, T. Breitkreutz and S. Sutphen, NMP—A Network Multi-processor, TR88-22, Computing Science Dept., Univ. of Alberta, Dec. 1988.

[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.

[Sec86] S. 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. of the ACM* **28(1)**, 22-33 (Jan. 1985).

Appendix: Prime Number Generator Example Program

The following program demonstrates the use of the following routines in a distributed prime number generator: `NodeInit`, `GetNodes`, `NodeClose`, `IsConnected`, `SendNode`, `RecvNode`, `PollAll`, and `ClearMask`. First, the sequential version is given:

```
/*
 * Prime number generator
 *
 * Usage: prime start finish
 * where start and finish are natural numbers.
 */


# include <stdio.h>
# include <math.h>

    /* function prime:  returns 1 if n is prime,
                  0 otherwise */
static  int
prime(n)
    int n;
{
    int a, sn;
    double  sq;
    extern  double  sqrt();

    if (n < 1)
        return 0;
    if (n < 4)             /* 1, 2, and 3 are prime */
        return 1;
    sq = sqrt((double) n);
    sn = (int) sq;
    if (((double) sn) == sq)    /* not prime if there is a perfect SQRT */
        return 0;
    for (a = 2; a <= sn; a++)
        if ((n % a) == 0)
            return 0;
    return 1;
}

int
main(argc, argv)
    int argc;
    char    **argv;
{
    int start, finish, a, n, cols;
    char    fmt[8];

    /* Check arguments. */
    if (argc != 3) {
        fprintf(stderr, "Usage: prime start end\n");
        return -1;
    }
    if (((start = atoi(argv[1])) < 0) || ((finish = atoi(argv[2])) < 1)) {
        fprintf(stderr, "Illegal start or end parameters\n");
        return -1;
    }

    /* Set up output format. */

    a = 2 + (int) log10((double) finish);
    sprintf(fmt, "%%%dd", a);
    cols = (79-a)/a;

    printf("Prime numbers from %d to %d:\n", start, finish);

    /* Loop through range. */

    for (n = 0, a = start; a <= finish; a++) {
        if (prime(a)) {
            printf(fmt, a);
            if (n++ >= cols) {
                printf("\n");
                n = 0;
            }
        }
    }
    if (n)
        printf("\n");
    exit(0);
}
```

Figure A-1. Prime Number Generator: Uniprocessor Version

The `prime` function simply checks for a prime number. `main` checks its range arguments, and simply loops through the range given checking for primes.

To write the NMP version, it is split into two programs, the *root* and the *helper*:

```
/*
 * NMPrime:  Root part
 * Sends start and finish parameters and collects results from helpers.
 * Each helper must be connected to the root in the configuration file.
 */
# include <stdio.h>
# include <math.h>

int
main(argc, argv)
    int argc;
    char    **argv;
{
    int start, finish, a, b, n, nodes, cols;
    char    fmt[8];
    int chunksize, msg[2];
    int pollr, Helpers[16];

    /* Check arguments. */

    if (argc != 4) {
        fprintf(stderr, "Usage: prime config start end\n");
        exit(-1);
    }

    if (((start = atoi(argv[2])) < 0) || ((finish = atoi(argv[3])) < 1)) {
        fprintf(stderr, "Illegal start and end parameters\n");
        exit(-1);
    }

    /* Start the NMP. */

    if ((nodes = NodeInit(0, argv[1])) < 0) {
        fprintf(stderr, "unsuccessful return from NodeInit!\n");
        exit(-1);
    }

    /* Send out node parameters. */

    chunksize = (finish-start) / (nodes-1);

    for (a = 1; a < nodes; a++) {
        if (!IsConnected(a)) {
            fprintf(stderr, "Root must be connected to all nodes.\n");
            NodeClose();
            exit(-1);
        }
        printf("Node %d (%s) does %d to %d.\n", a, GetAnyHost(a),
            start+(a-1)*chunksize,
            (a == nodes-1) ? finish: start+a*chunksize-1);

        /* Send the range for each node to do. */

        msg[0] = htonl(start+(a-1)*chunksize);
        msg[1] = (a == nodes-1) ? htonl(finish) : htonl(start+(a)*chunksize-1);
        if (SendNode(a, (char *)msg, sizeof(msg)) != sizeof(msg)) {
            fprintf(stderr, "Unable to send.\n");
            NodeClose();
            exit(-1);
        }
    }

    /* Set up output format */

    a = 2 + (int) log10((double) finish);
    sprintf(fmt, "%%%dd", a);
    cols = (79-a) / a;

    printf("Prime numbers from %d to %d:\n", start, finish);


    /* Poll for results. */

    ClearMask(0);    /* Exclude polling of root */

    n = 0;
    while (nodes > 1 && pollr = PollAll(Helpers, 1)) {
        for (a = 0; a < pollr; a++) {

            /* Receive pending messages. */

            if (RecvNode(Helpers[a], (char *)msg, sizeof(int)) != sizeof(int)) {
                fprintf(stderr, "Unable to receive from %d.\n", Helpers[a]);
                NodeClose();
                exit(-1);
            }


            b = ntohl(msg[0]);
            if (b == -1) {

            /* The node is done. */

                nodes--;
                ClearMask(Helpers[a]);

                /* Acknowledge */
```

```
                if (SendNode(Helpers[a], msg, 1) != 1) {
                    fprintf(stderr, "Unable to send.\n");
                    NodeClose();
                    exit(-1);
                }
            }
            else {

            /* Print out the prime. */
                printf(fmt, b);
                if (n++ >= cols) {
                    printf("\n");
                    n = 0;
                }
            }
        }
    }
    if (n)
        printf("\n");

    /* Terminate the NMP session. */

    NodeClose();
    exit(0);
}
```

Figure A-2. Distributed Prime Number Generator: Root

The root node does not check for any primes itself, but it initiates the NMP with a call to `NodeInit`, it divides the range of numbers to check into equal sized chunks, and it distributes them. The last helper node is a special case and gets whatever sized chunk is remaining. Then, after the output formatting is set up, the root simply waits for results to come in from the helpers, using `PollNode`. Completed nodes are detected by a negative number received, and are eliminated from polling by a call to `ClearMask`. A single byte acknowledgement is sent to finished helpers to ensure the communication paths are flushed before any helpers exit. The prime numbers are printed out one at a time as they are received from the helpers.

```
/*
 * NMPrime:  Helper part
 * Receives start and finish parameters, calculates primes, and sends prime
 * values back to root.  Each helper must be connected to the root in the
 * configuration file.
 */
# include <stdio.h>
# include <math.h>

/* Prime number checker. */

static  int
prime(n)
    int n;
{
    int     a, sn;
    double  sq;
    extern  double  sqrt();

    if (n < 1)
        return 0;
    if (n < 4)             /* 1, 2, and 3 are prime */
        return 1;
    sq = sqrt((double) n);
    sn = (int) sq;
    if (((double) sn) == sq)    /* not prime if it has perfect SQRT */
        return 0;
    for (a = 2; a <= sn; a++)
        if ((n % a) == 0)
            return 0;
    return 1;
}

int
main(argc, argv)
    int argc;
    char    **argv;
{
    int start, finish, a, n, cols, msg[2];

    /* Initialise NMP Node. */

    if (NodeInit(1, "") < 0) {
        fprintf(stderr, "unsuccessful return from NodeInit!\n");
        exit(-1);
    }

    /* Make sure we are connected to the root node. */

    if (!IsConnected(0)) {
```

```
    fprintf(stderr, "Not connected to root.\n");
    NodeClose();
    exit(-1);
}

/* Get our startup parameters. */

if (RecvNode(0, (char *)msg, sizeof(msg)) != sizeof(msg)) {
    fprintf(stderr, "unsuccessful return from RecvNode!\n");
    NodeClose();
    exit(-1);
}
start = ntohl(msg[0]);
finish = ntohl(msg[1]);

/* Loop through our assigned range. */

n = 0;
for (a = start; a <= finish; a++)
    if (prime(a)) {

        /* Put in network format, send to root. */

        msg[0] = htonl(a);
        if (SendNode(0, (char *)msg, sizeof(int)) != sizeof(int)) {
            fprintf(stderr, "Unable to send.\n");
            NodeClose();
            exit(-1);
        }
    }

/* Tell Root we are done. */

msg[0] = htonl(-1);
if (SendNode(0, (char *)msg, sizeof(int)) != sizeof(int)) {
    fprintf(stderr, "Unable to send.\n");
    NodeClose();
    exit(-1);
}

if (RecvNode(0, (char *)msg, 1) != 1) {
    fprintf(stderr, "Unable to receive acknowledgement.\n");
    NodeClose();
    exit(-1);
}

/* Terminate this node. */

NodeClose();
exit(0);
```

```
}
```

Figure A-3. Distributed Prime Number Generator: Helper

The helper `prime` function is exactly the same as the sequential program's. The helper begins with a `NodeInit` to receive its startup information and the interconnection matrix. It then makes sure it is connected to the root node, in case the user has given a bad configuration file. It receives its startup parameters from the root, and loops through its range checking for primes. When it detects a prime number, it sends it to the root, one prime per message. When finished, it sends a negative number to the root, then waits for the single byte acknowledgement, and finally terminates.

Index of Functions