Proceedings of the 1988 Winter Simulation Conference
M. Abrams, P. Haigh, and J. Comfort (eds.)

A simulator for asynchronous hypercube communications

M. G. Roth
Mathematical Sciences Department
University of Alaska-Fairbanks
Fairbanks, AK 99775

and

M. D. Wiley
Naval Research Laboratory
Washington, DC

ABSTRACT

This paper describes a simulator for the development and study of software for hypercube multiprocessors with asynchronous communications. The main area of application for the simulator is the development of non-numerical algorithms for hypercubes. The paper discusses the properties of hypercube communications networks, the programmer interface to the simulator, the simulator implementation and programming examples.

1. INTRODUCTION

The program described in this paper simulates the communications system used on a hypercube multiprocessor under development at the Jet Propulsion Laboratory (Core, 1985). The simulator was designed to aid in developing programs for that system and to study the asynchronous communication traffic patterns which occur in hypercube multiprocessors in general. The simulator runs under Version 4.4 or later of the VAX/VMS operating system and may be used with any other software which uses the VAX/VMS Common Language Environment.

The simulator is capable of running up to a 6-dimensional (64 node) hypercube on a VAX 11/750 or larger machine. We refer to each simulated processor as a *node*. It is assumed that the hypercube communicates to the outside world through an intermediate processor called a *Control Processor* (CP). Please note that there is vitually no similarity between this simulator and an earlier synchronous (Crystalline) simulator described by Hunt (1985), although the synchronous system provides a subset of the functional capabilities in the present simulator.

The simulator allows any node to communicate with any other node or with the CP. It also allows the CP to broadcast messages to all nodes. Each node maintains a queue of messages received from the other nodes and the CP. These messages can be accessed in FIFO order on messages from all nodes or on only those messages from a specific node. In addition, the handling of variable size messages is transparent to the user.

The asynchronous simulator was developed primarily for the purpose of investigating non-numerical algorithms on the hypercube architecture. A wide variety of numerical problems have been programmed and run successfully on hypercubes. The majority of the numerical applications use regular spatial decompositions in which identical computations are performed on disjoint subsets of a lattice in the physical solution space. Such applications, known as homogeneous problems, are programmed by loading each hypercube processor with an identical program to perform computations on a subset of the lattice. For such applications, synchronous communication between processors is usually sufficient because the computations at each processor are more or less identical. Since a processor waiting for input blocks until a message is received, the effect of synchronous communication is to synchronize the computations of all processors by means of their input requests.

If the amount of computation between corresponding input requests is equal in every node, the processor loads will be perfectly balanced and, assuming that the CPU execution time is linear in the number of points of the physical lattice, a speedup equal to

$$(2^N/(1+C/E)) \tag{1}$$

is achieved, where E is the CPU execution time for a single node processor to complete a solution on a subset of the lattice, and C is the communication time required for the same solution. When C is small relative to E, the maximum speedup of 2^N is nearly achieved.

In non-numerical applications, such as logic programming, processor load balancing is much more difficult than for numerical problems. For numerical problems it is usually possible to develop an error estimator for the solution which determines the step sizes and grid spacings required in the calculations. From this information, it is possible, at least in theory, to subdivide a given problem in such a way that the processor loads are approximately balanced.

In logic programming, there is often no obvious way to subdivide the problem into equal pieces. Thus, a more general scheduling strategy is required. The approach adopted here, and implemented in the hypercube simulator is that each node processor runs a copy of the same program, which is typically a Prolog or Lisp interpreter. However, the input to the interpreter for each processor constitutes a different program. In effect, each processor therefore executes a different program. In this environment, processor loading can be adjusted dynamically only if asynchronous communication between nodes is possible. This capability will be available in the 32 node hypercube currently under development at JPL and has been implemented in the simulator described herein.

2. HYPERCUBE MODEL

The multiprocessor architecture known as the Hypercube or Nearest Neighbor Concurrent Processor consists of 2^N individual processors interconnected in the same manner that the vertices of a binary N-dimensional hypercube are connected by its edges. Each processor contains private memory and may or may not have access to shared memory or shared file systems, depending on the implementation. A hypercube is typically controlled by a control processor (CP), which communicates directly with one or more of the node processors. In the Caltech/JPL hypercubes, the CP communicates directly with only a single node of the hypercube.

The hypercube has interesting properties which make it useful as the basis for communications networks. One of these is that for an exponential growth in the number of nodes there is only a linear growth in the number of communications links required for a given node. This makes it possible for the hypercube to grow rapidly in its computational power with a relatively small communications cost increase per node.

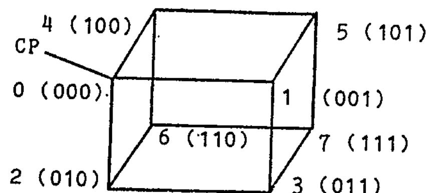
In addition, the model allows the use of a binary numbering scheme which allows for a fast message routing algorithm and guarantees a minimal length path. This last attribute is extremely important since we want the communications overhead to be as small as possible. This binary representation is also useful for theoretical analysis.

Finally, the hypercube includes as special cases other common interconnection topologies such as rings, trees, and toroids.

2.1 Representation And Connectivity

A formal hypercube definition is presented in Section 2.2 and is very useful in the theoretical sense, but fails to give an intuitive understanding of how the hypercube may be represented physically. The most common representation of a 3-cube is shown in Figure 1. Each vertex of the cube corresponds to a node whose number is shown in decimal and (binary) next to the vertex. Each vertex in an N-dimensional cube is directly connected to N neighbors. Also shown in Figure 1 is the CP, which communicates with the hypercube through node 0.

Figure 1. Three dimensional hypercube and control processor.



Note that the neighbors of a given node n have numbers which differ from n by exactly 2^j for $j = 0, \ldots, N-1$, where N is the order of the cube. This coding system is called "Gray code" and is particularly useful for message routing.

2.2 N-Dimensional Cube Construction

The connectivity graph and node numbers in an N-dimensional hypercube can be generated by a recursive algorithm. The following algorithm describes the generation of an N-dimen-sional hypercube in a form which is particularly useful for the analysis of message flow within the hypercube.

1. Let C_N be a hypercube of N dimensions (2^N processors).
2. Define C_0 to consist of node $I_0 \leftarrow 0$ only, with no connections.
3. To construct C_N, take two identical cubes C_{N-1} and C'_{N-1} and connect them as follows:
   - 3.1 For each node I_j in C_{N-1}
     - 3.1.1 Connect node I_j to the corresponding node I'_j in C'_{N-1}
     - 3.1.2 Rename I'_j to $I_{j+2^{N-1}} \leftarrow j + 2^{N-1}$

2.3 Routing Algorithm

Each node has I/O channels numbered $0, \ldots, N-1$. The channel by which two directly connected nodes communicate is numbered according to the position of the bits in which the numbers differ, where position 0 is the least significant bit. For example, nodes 0 and 1 differ in the $0th$ bit position, thus they are connected via channel 0. Messages are passed according to the following algorithm:

1. Let x = position of the lowest bit set in Current_Node XOR Destination_Node. (Bit numbering is $N-1,\dots,0$.)
2. If the Current_Node = Destination_Node
   Then Save the message.
   Else Send the message out channel x.

2.4 Message Routing

Section 2.3 described how the message routing works from the point of view of a given node. It is also useful to know the path a message takes when going from the source node to the destination node. The following algorithm generates the list of nodes visited.

1. Let I_0 = Source_Node and D = Destination_Node
2. $j \leftarrow n \leftarrow 0$
3. While $j < N$ Do
   3.1 If bit j of (I_n XOR D) = 1 Then
       Add node $I_{n+1} \leftarrow (D \ \& \ 2^j) \mid (I_n \ \& \ \sim 2^j)$
       and $n \leftarrow n+1$
   3.2 $j \leftarrow j+1$

The term $(D \ \& \ 2^j)$ masks the destination address. The term $(I_n \ \& \sim 2^j)$ masks all bits except the the jth from the current node in the list. The result of ORing these terms is to shift one bit from the source node's value to the destination node's value. Each time this is done the next node closer to the destination is generated. The list $I_0,\dots,I_n$ represents the path a message must travel to go from the source to the destination.

2.5 Worst Case Analysis

Well designed hypercube programs take advantage of "locality of reference" to reduce communications overhead. As mentioned above this is not always possible in applications where it is difficult to predict the course of the solution. It is therefore useful to analyze the case where every node sends a message to every other node. This causes the worst communications problems possible due to the large number of packets each node handles. We analyzed the number of packets handled by each node to determine the communications overhead in this case. Section 5 describes the programs used to study this example on the simulator.

2.5.1 Connectivity. Node I is directly connected by a path of length 1 to the nodes which satisfy

$$(I \text{ XOR } 2^n),\ n = 0,\dots,N-1. \tag{2}$$

Proof: This follows directly from the method of construction.

2.5.2 Path Length. A message from node I to node J passes through

$$K = \sum_{\text{bits}} (I \text{ XOR } J) \tag{3}$$

nodes.

Proof: This follows directly from the routing algorithm since each bit in (I XOR J) is looked at exactly once. Only if a given bit is set is the message sent from the current node to the next node closer to J.

2.5.3 Number Of Messages. The number of messages, M_{N-1}, either ending at or passing through every node in an N-cube for the case where all nodes are sending exactly one message to all other nodes is

$$M_N = N \times 2^{N-1} \text{ for } N > 0. \tag{4}$$

Proof: Using the N-cube definition given previously, the number of messages reaching but not necessarily stopping at a given node I is

$$M_N = 2M_{N-1} + 2^{N-1}. \tag{5}$$

This can be shown by noting that there are M_{N-1} messages at I due to the $N-1$ order cube C, 2^{N-1} messages from I' to C via I, and M_{N-1} messages from all nodes in C' other than I'. By induction it can be shown that this recurrence equation reduces to (4).

The number of messages passed for a hypercube consisting of a single node ($N = 0$) is $M_0 = 0$. Now assume $M_{N-1} = (N-1) \times 2^{N-2}$. Then from (5)

$$\begin{aligned} M_N &= 2M_{N-1} + 2^{N-1} \\ &= 2((N-1) \times 2^{N-2}) + 2^{N-1} \\ &= (N-1) \times 2^{N-1} + 2^{N-1} \\ &= N \times 2^{N-1} \end{aligned} \tag{6}$$

3. PROGRAMMER INTERFACE

The user must provide two programs for the simulator. One is the program for the Control Processor (CP), which is connected to the hypercube as shown in Figure 1. The other

program is loaded into every node of the hypercube. These programs access the hypercube simulation features via a set of functions and subroutines whose names begin with "HA_". Figure 2 shows a list of the simulator routines with a brief description of each. The declarations for these routines are normally included in a header file at the beginning of the CP and node programs.

Figure 2. C Language declarations file for hypercube simulator.

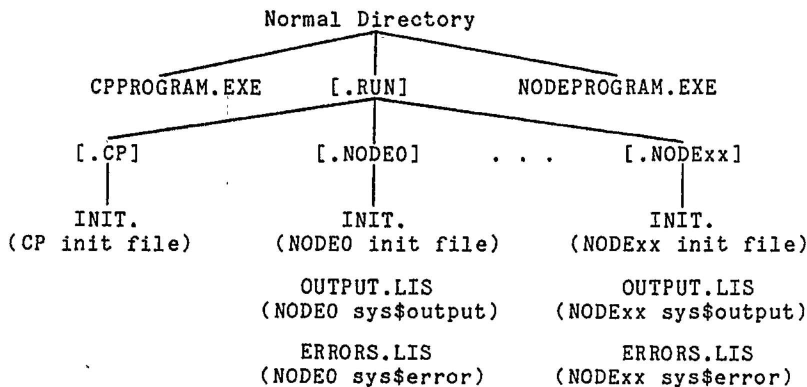
```
/* CP initialization routine                */
extern int ha_init_CP();

/* Node initialization routine              */
extern int ha_init_node();

/* Get a message from node n or the CP      */
extern int ha_read_node();

/* Send a message to node n or the CP       */
extern int ha_write_node();

/* CP broadcasts a message to all nodes     */
extern int ha_broadcast();

/* Get # messages waiting from node n       */
extern int ha_status();

/* Gracefully shut cube down (Normal)       */
extern int ha_shutdown();

/* Fatal error so shut down the cube        */
extern int ha_error();

/* Get global simulator values              */
extern int ha_whoami();

/* Communication overhead summary           */
extern int ha_summary();
```

Although the simulator routines are written in C, they may be called from CP or node programs written in any language which uses the VAX/VMS Common Language Environment. When programming in C, the header file is called HASIM.H. To program in other languages the declarations in HASIM.H must be translated into the appropriate format before they can be included in the CP and node programs. Header files for FORTRAN and Pascal are presently available. The reader is referred to Wiley (1986) for additional information on mixed language simulations.

The low level message passing is invisible to the programmer and thus no such routines are listed. The use of certain routines is required for successful execution of the software. For example, the routines HA_INIT_CP and HA_INIT_NODE initialize the CP and nodes, respectively. HA_SHUTDOWN is also required to provide an orderly termination of the simulated processes. The remaining routines are used for sending and receiving messages, obtaining information about individual nodes and statistics about the the communications at each node.

The programmer is provided with the simulator source code file, the include files used to provide the simulator subroutine definitions and two VAX/VMS Command Language procedures, HACONF.COM and BUILDDIR.COM, which are used to interactively obtain information on cube order and mailbox size and to create the necessary directory structure to run a simulation. The organization of these directories and their contents are shown in Figure 3. The CP and node programs reside in the top level directory, which also contains a subdirectory named RUN. In this subdirectory each of the node processors and the CP have a subdirectory to which their output is directed.

Figure 3. Hypercube simulator directory structure.



4. SIMULATOR IMPLEMENTATION

The simulator operates by creating a VMS subprocess for each node. The original process becomes the CP. The CP and each node process has a mailbox to which all messages for that process are sent. All processes run in separate sub-directories, with node output going to files in the subdirectory owned by the node and CP output going to the user's default output device (i.e. terminal). In addition, each node (except the CP) also has an event flag associated with it, which is used to provide synchronization during startup and shutdown. Figure 4 shows the I/O diagram for a given node. Several VMS System Services are used to handle low level communications and synchronization. Figure 5 gives an overview of what happens in the simulator during startup and shutdown and is discussed in more detail in the following sections.

4.1 Initialization

The CP program reads the data file generated by HACONF.COM, which contains data regarding the order of the hypercube, the mailbox size, and the name of the node program. The event flag clusters used for hypercube synchronization during startup and shutdown are allocated and all event flags are cleared. The CP then creates a mailbox for itself and one for each node. Initialization files containing mailbox and neighbor information are written to each node's directory and the node processes are created. When created, each node's I/O is redirected to files within its own sub-directory and the process name and system limits are also set. When the CP is done with these tasks it waits until all nodes report via event flags that they have completed their initialization.

When created, each node program must complete its own initialization process via calls to HA_INIT_NODE. It must read the data file generated by the CP, assign channels to its own and all neighbor's mailboxes, and initialize its own message queue. When these operations have been completed, it sets its event flag and hibernates until the CP wakes it.

When all nodes have reported then the CP again clears all common event flags (for use later by the shutdown routine) and wakes all the nodes. The hypercube simulator is now ready for operation.

4.2 Message Passing

All messages are passed using fixed size packets via VMS mailboxes and asynchronous system I/O calls. There is a mailbox associated with each process (CP and node) which interrupts the process whenever a message is received. When the process is interrupted it keeps and/or passes along the received message as required. Thus the user program does not have to worry about the mechanics of the message passing system.

It is common to have a message which fills more than one packet. In this case the packets are stored normally, but the available message count for a given node is not updated until a packet containing at least one null character is received. If the message would fill exactly n packets, then $n+1$ packets are sent, with the last packet being empty. Only when such a packet is received does the queue manager acknowledge that the message is available.

If a process requests a message which is not available, the process hibernates until the queue manager signals that a message is ready. If the new message is not the one desired then the process hibernates repeatedly until the correct message is available.

4.2.1 Message Routing. Message routing is done using the following algorithm:

Let $X =$ (Current_Node XOR Destination),
    where Destination = 0 for the CP.
If the message type = B (Broadcast mode) Then
    Keep a copy of the message; and
    For $n = 0$ to this node's subcube order -1
        Forward a copy of the message out channel n.
Else If the (Current_Node = Destination) Then
    Save the message.
Else If this node is the CP Then
    Send the message via the CP channel to node 0.
Else If the Destination is the CP Then
    If Current_Node $\neq$ node 0 Then
        Send the message towards node 0.
    Else Send the message to the CP using
        the CP channel from node 0.
Else Send the message out channel X.

4.2.2 Message Queue Structure. All messages received by a node are stored in a global message queue for that node. Each node maintains two arrays of pointers into its global message queue. These pointers are used to directly access messages from a given node. The first array, FRSTMSG, points to the first packet in the queue from a given node. The second array, LSTPACK, points to the last packet from a given node. These pointers allow the front and back of the queue to be located without list traversal. Additional pointers give the HEAD, TAIL, and next FREE blocks in the queue.

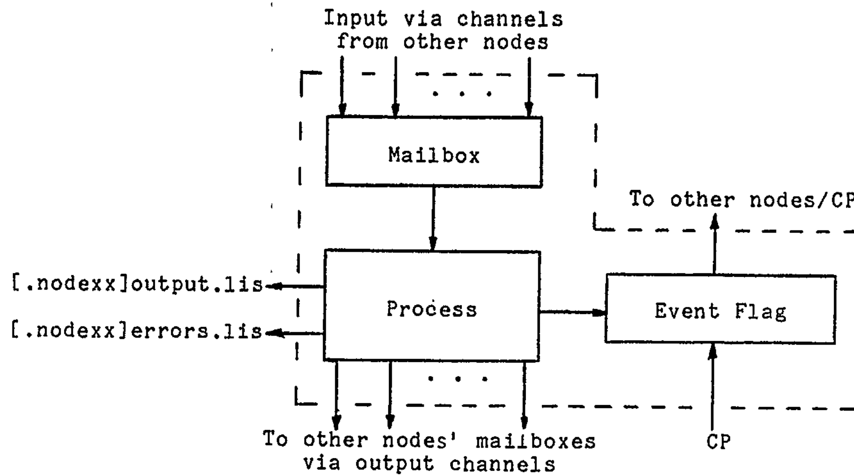


Figure 4. I/O organization for a simulated hypercube node processor.

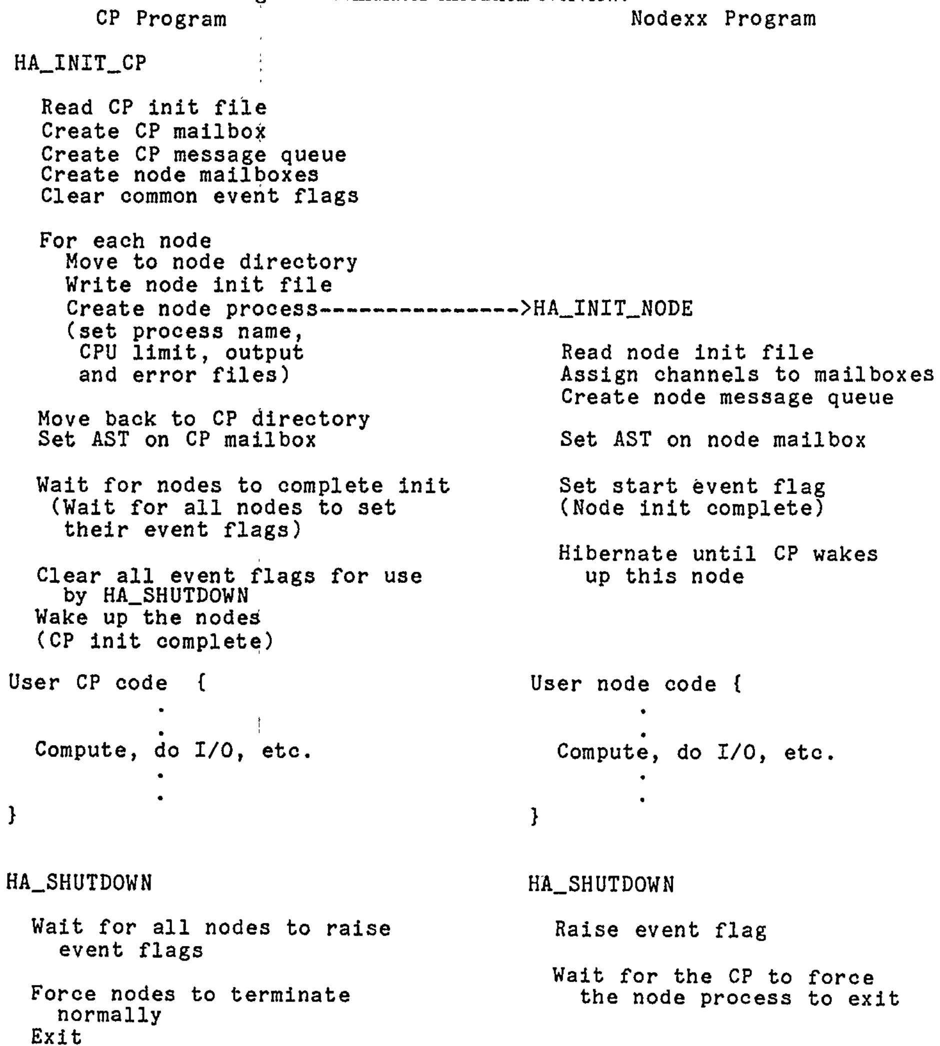


Figure 5. Simulator execution overview.

In addition to the message packets, the global queue contains a set of links enabling it to keep data about message sequencing, source nodes, and total message length. Figure 6 demonstrates what the queue for a 3-dimensional cube might look like. The specific node numbers used in the figure are for demonstration purposes only.

The queue, arrays and global queue pointers for an N-dimensional cube are defined below:

```
struct inqueue {
   int *next;     next packet from same node
                   or next free block
   int *next_msg; next packet in global queue
   int *prev_msg; back link in global queue
   char srcnode;  where packet came from
   char bytcnt;   number of bytes used in packet
   char data[x];} data (x=Packetsize-Packet_Hdr)

struct inqueue *frstmsg[N]; head of node queues
struct inqueue *lstpack[N]; tail of node queues
```

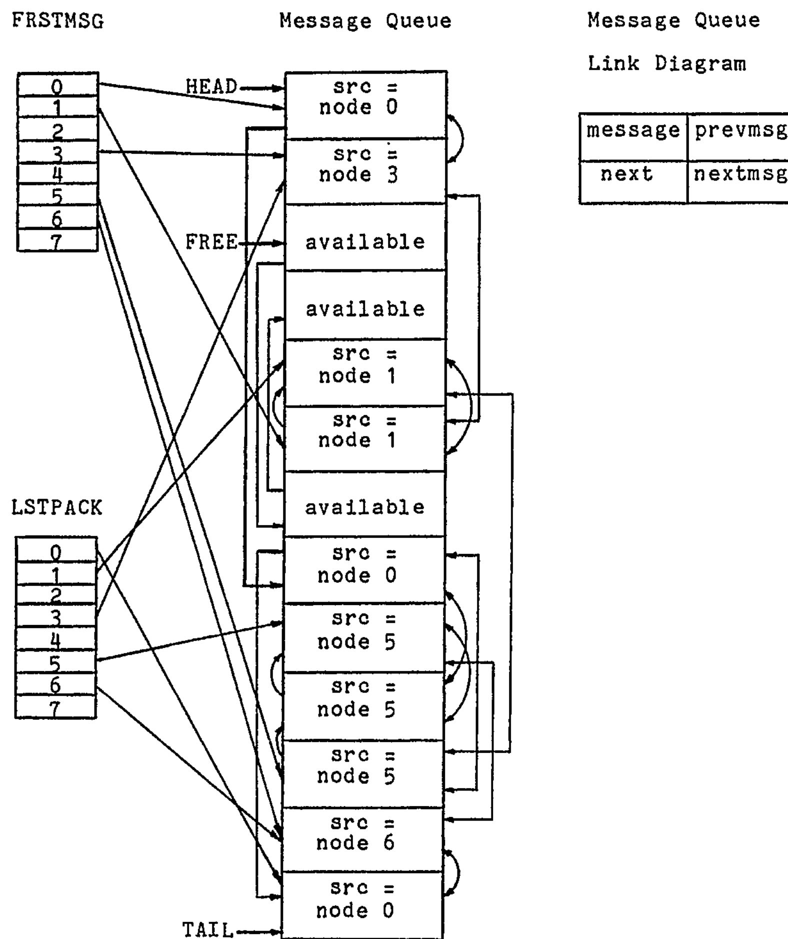
```
struct inqueue *free;  head of free list
struct inqueue *head;  head of global queue
struct inqueue *tail;  tail of global queue
```

4.2.3 Packet Format. Messages are sent between nodes in 16 byte packets by default. The packet length may be changed by changing the value for Packetsize in the simulator source code and recompiling. The packet format is as follows:

```
byte  0      : Message type
               B - broadcast
               D - data
               E - error
byte  1      : Destination node
byte  2      : Source node
byte  3      : Number of data area bytes used
bytes 4 - 15 : Data area (Packetsize-Packet_Hdr)
```

4.2.4 I/O Routines. The CP may send messages to the nodes via either of two methods. It may broadcast messages to all nodes via HA_BROADCAST or it may send a message to

Figure 6. Message queue data structure for 3-D hypercube.



a specific node via HA_WRITE_NODE. The nodes may only use the latter method to send messages. They may be read, via HA_READ_NODE, either with respect to the global order in which messages arrived or by specifying a particular node.

4.3 System Calls

VAX/VMS provides a variety of system services which were useful in writing the simulator. In particular, the $QIO (Queue I/O Request) was critical to the ease with which this software was implemented. This service provides the ability to deliver messages to a mailbox synchronously and to read them asynchronously. Since the sending operation is synchronous, it will wait until the message is delivered before completing. Thus if the destination mailbox is full then it will wait until the receiving node has had time to process a few messages and partially empty its mailbox.

Conversely, the asynchronous reception of messages allows the receiving node to process the incoming messages without attention by the user program. Thus, this mechanism is completely transparent to the user. This is especially important when a message is received which is not for the current node and must be passed on.

Unfortunately, this method does not avoid the possibility of deadlock caused by mailbox overflow. However, if mailbox sizes are sufficiently large then this does not become a problem. Evaluation of the worst case test problem discussed in Section 2.5 indicates that the minimum mailbox sizes provided by HACONF.COM are sufficiently large for most applications.

4.4 Run Statistics

The routine named HA_SUMMARY writes runtime statistics to the default output device. These data include CPU time, number of bytes, packets, and messages sent, received, or forwarded, and the average packet and message lengths. This routine may be run at any time to obtain intermediate as well as final results.

The number of bytes sent are those which this node or CP originated. The number received are those for which this node was the destination. The number forwarded are those which were neither sent or received, but were simply passed along. The number broadcast is the number of bytes received (sent for the CP) that had a message type of "Broadcast." These broadcast values are not reflected in the sent/received/forwarded information given earlier and are completely independent. The packet and message information is generated in a similar manner.

4.5 Error Handling

A special purpose routine, HA_ERROR, is provided to handle fatal errors. In the CP this routine will print a message to the error device. Using the process identifications (PIDs) saved when creating the nodes, it will force all of the nodes to terminate. It will then shut itself down and return the exit status.

When a node calls this routine a special type of message is sent to the CP which then calls HA_ERROR. The error message printed by the CP identifies which node originated the message. In addition, the error status sent from the node is used as the CP exit status. This allows VMS to interpret the error status and let the user know what it means. Information indicating where the error occured is usually written to the node's error file.

4.6 Shut Down Sequence

A routine called HA_SHUTDOWN is provided which allows for the orderly shutdown of the simulator after the program has completed. Although one node may be finished with its portion of the processing, because of the asynchronous nature of the simulator it may be required to pass messages between nodes which have not completed work. Thus it may not simply shut itself down. This procedure avoids all of the potential problems associated with shutting down the simulator.

When HA_SHUTDOWN is called a node sets its event flag and waits. This wait does not interfere with message handling but does reduce the CPU overhead involved with running many processes. When the CP is done it waits for all of the nodes to raise their event flags, then it tells all of the node processes to exit normally.

5. ALL-TO-ALL SIMULATION

In this simulation the CP broadcasts a message to all nodes and also sends an individual message to each node. Each node also sends a message to all other nodes and the CP. After sending the appropriate messages, each node/CP waits until all expected messages have been received from the other nodes/CP, then prints the communications summary and stops. These programs use almost all of the functions available on the simulator and provide a good benchmark on the highest possible communication traffic.

The C language source codes for the CP and node programs for a 3 dimensional hypercube simulation (8 processors) are given in Appendix A. Ignoring messages from the CP, each node will receive or forward $M_3 = N \times 2^{N-1}$ messages, according to (4). Thus, for $N = 3$, $M_3 = 12$. The communications summaries for node 0 and node 7 are shown in Figure 7. The summaries confirm that the correct numbers of received and forwarded messages are obtained when the CP messages are subtracted. At node 0, 14 CP messages are forwarded (7 from the CP to the other nodes and 7 from the other nodes to the CP) and 1 CP message is received. In the case of node 7, only 1 CP message is received and no messages are forwarded to other nodes.

one node found a solution to a problem on which several nodes were working via different approaches. The first node to succeed would send an answer to the controlling node/CP which would then send "stop" messages to the other nodes working on the problem. The implementation of this capability would be analogous to the way in which messages from different nodes are now accessed in the queues.

6.2 Removal Of The 64-node Limit

The simulator is limited to 64 nodes due to the startup and shutdown synchronization method, which employs event flags. VAX/VMS allows a maximum of 64 common event flags to be used by communicating processes. A possible solution to this problem would be to lock global pages into memory which could then be used by all processes to perform the same synchronization tasks. Unfortunately, this also requires special operating system privileges and has the ability to degrade the CPU performance if done incorrectly.

Figure 7. Communications summaries for "All-to-All" simulation.

```
Communication summary: Node 0

 # of          Sent       Broadcast     Forwarded     Received
---------------------------------------------------------------
Bytes           48            20            177           57
Packets          8             2             26            9
Messages         8             1             19            8

Average packet size = 6.7 bytes
Average message size = 1.3 Packets, 8.4 bytes
CPU time used = 1.15 seconds


Communication summary: Node 7

 # of          Sent       Broadcast     Forwarded     Received
---------------------------------------------------------------
Bytes           48            20             30           57
Packets          8             2              5            9
Messages         8             1              5            8

Average packet size = 6.5 bytes
Average message size = 1.1 Packets, 6.0 bytes
CPU time used = 1.17 seconds
```

6. FUTURE DEVELOPMENTS

There are a few modifications which may be implemented at a future date which would expand the capabilites of the simulator. Some of these changes are due to a better understanding of the capabilites required for the physical hardware on which this software is based.

6.1 Prioritized Messages

It would be useful in certain applications (e.g. Concurrent Prolog) to have a mechanism for sending messages of different priorities. This would be particulary useful when telling a node to stop work on the current problem. This could happen when

6.3 Multiple Node Programs

It may be desirable in some cases to have the nodes running different programs. For example, this could happen when learning to partition the hypercube into subcubes to take advantage of unneeded processors. Currently only copies of a single program may run in each of the nodes. This limitation

could easily be removed to allow different programs to run in each node by having HACONF.COM either request the name of a load map file or interactively querying the user for the program to run in a specific node. HA_INIT_CP would also have to be modified to read the load map and use the specified programs when starting the nodes.

6.4 Packet Length

In some cases a simpler mechanism for specifying packet length may be desired. For example, a user may run a series of simulations in which the optimum packet size varies greatly. In this case it would be useful to have HACONF.COM query the user for the packet size desired. This would be easy to implement, requiring only small changes in HACONF.COM, HA_INIT_CP, HA_WRITE_INIT, and HA_READ_INIT.

ACKNOWLEDGEMENTS

This work was partially funded by NASA through JPL contract 956901.

APPENDIX A. All-to-All Programs

These programs illustrate the worst case communication traffic in which every node sends messages to every other node as discussed in Section 5.

A.1 All-to-All CP Program

```
#include "hasim.h"

main()
{
  char    buffer[30];
  int     i,j,msgnode,length;
  int     nodenum,doc,dim,numnodes;
  int     trace,pid;

  /* Initialize CP */
  ha_init_CP();
  ha_whoami(&nodenum,&doc,&dim,
          &numnodes,&trace,&pid);

  /* Broadcast message to all nodes */
  sprintf(buffer,"Broadcasting message");
  printf("Broadcast message\n\n");
  ha_broadcast(buffer,20);

  /* Send message to all nodes */
  sprintf(buffer,"CP sending data");
  for (i = 0; i < numnodes; i++)
  { printf("Sent message to node %d\n",i);
    ha_write_node(i,buffer,15);
  }
  /* Read messages from all nodes */
  printf("\n");
  for (i = 0; i < numnodes; i++)
  { printf("\nWaiting for message %d\n",i);
    length = ha_read_node(nodenum,&msgnode,
                          buffer, 20);
    buffer[length] = '\0';
    printf("Message received from node %d,
            length is %d\n",msgnode,length);
    printf("Message is '%s'\n",buffer);
  }
  /* Get communications summary */
  ha_summary();

  /* Shut down simulator */
  ha_shutdown();
}
```

A.2 All-to-All Node Program

```
#include "hasim.h"

main()
{
  char    buffer[20];
  int     i,j,length,dest,msgnode;
  int     nodenum,doc,dim,numnodes;
  int     trace,pid;

  /* Initialize node processor */
  ha_init_node();
  ha_whoami(&nodenum,&doc,&dim,
          &numnodes,&trace,&pid);

  /* Send a message to all other nodes */
  sprintf(buffer,"Node %d",nodenum);
  for ( i = 0; i < numnodes ; i++ )
  { if (i == nodenum) dest = -1;
    else dest = i;
    ha_write_node(dest,buffer,strlen(buffer));
    printf("Sent message to node %d\n", dest);
  }
  /* Receive messages from all other nodes */
  printf("\n");
  for ( i = 0; i< (numnodes + 1) ; i++ )
  { printf("Waiting for message %d\n", i);
    length = ha_read_node(nodenum, &msgnode,
                          buffer, 30);
    buffer[length] = '\0';
    printf("Message received from node %d,
            length is %d\n",msgnode, length);
    printf("Message is '%s'\n",buffer);
  }
  /* Get communications summary */
  ha_summary();
  /* Wait for CP to complete shut down */
  ha_shutdown();
}
```

REFERENCES

Core Engineering Team (1985). *Mark III Core Engineering Notebook*, JPL D-2431, Hypercube Research Project, Jet Propulsion Laboratory.

Hunt, H. (1985). "Hypercube Simulator: Nsim," Caltech Concurrent Computation Program, Hm-155.

Wiley, M. (1986). "A VAX/VMS Simulator for an Asynchronous Hypercube Architecture," Unpublished M.S. Thesis, Department of Mathematical Sciences, University of Alaska Fairbanks.

AUTHOR'S BIOGRAPHIES

MITCHELL G. ROTH holds a Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign. He is presently an associate professor of computer science at the University of Alaska in Fairbanks.

Department of Mathematical Sciences, University of Alaska, Fairbanks, Alaska 99775. (907)-474-7332.

MARSHALL D. WILEY earned the M.S. degree in computer science at the University of Alaska Fairbanks. He is presently a systems analyst for the Naval Research Laboratory.

Naval Research Laboratory, Washington, D.C. (206)-767-1009.