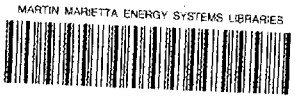

ORNL/TM-10410

ornl

OAK RIDGE
NATIONAL
LABORATORY

MARTIN MARIETTA

A Portable Hypercube Simulator

T. H. Dunigan

OPERATED BY
MARTIN MARIETTA ENERGY SYSTEMS, INC.
FOR THE UNITED STATES
DEPARTMENT OF ENERGY

Printed in the United States of America. Available from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Road, Springfield, Virginia 22161
NTIS price codes—Printed Copy: A03; Microfiche A01

ORNL/TM-10410

Engineering Physics and Mathematics Division

Mathematical Sciences Section

A PORTABLE HYPERCUBE SIMULATOR

T. H. Dunigan

Date Published - July 1987

The work was supported by the
Applied Mathematical Sciences subprogram
of the Office of Energy Research,
U. S. Department of Energy

Prepared by the
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831
operated by
Martin Marietta Energy Systems, Inc.
for the
U.S. DEPARTMENT OF ENERGY
under Contract No. DE-AC05-84OR21400

Table of Contents

Abstract 1

1. Introduction 1

2. User's Guide 2

2.1 Command interface 2

2.2 Simulator subroutines 3

2.3 Trace file and post-processors 5

2.4 Sample session 6

2.5 FORTRAN interface 7

2.6 Debugging 8

2.7 Intel iPSC and the Simulator 9

3. Implementation 10

3.1 Portable implementation 10

3.2 Sequent implementation 11

References 13

Appendix A: Simulator Manual Pages 14

A Portable Hypercube Simulator

T. H. Dunigan

Mathematical Sciences Section
Engineering Physics and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831

ABSTRACT

The structure and use of a hypercube simulator that runs on most UNIX* systems is described. The simulator uses a library of message-passing routines and multiple user processes, written in either C or FORTRAN, to provide an environment for the development and testing of algorithms for hypercube parallel processors. The simulator produces a trace file that can be used for debugging, performance analysis, or graphical display. A version of the simulator that uses the shared memory of a Sequent multiprocessor is also described.

1. Introduction.

This report describes a hypercube simulator that will run on most UNIX computing systems. The simulator has been run on UNIX 4.x BSD and derivative systems (including DEC's Ultrix, Sequent Dynix, Sun, IBM RT, and Encore), XENIX 3.x (including XENIX for IBM PC/AT and Intel 310), and System V UNIX (including AT&T 3B2). The portable simulator enables our staff to develop and test applications for the Intel hypercube, and then port those applications from the simulator to the hypercube with no source code changes. The simulator uses UNIX processes and pipes to simulate the parallel computing environment of the hypercube.

The simulator is a supplement to our interpretive simulator [1]. The interpretive simulator provides detailed message passing simulation, including simulated message delays with tunable message-passing parameters. It permits simulation of hundreds of processes. However, the interpretive simulator contains machine-language dependencies on the DEC VAX architecture, and the application programs must be altered in order to port them between the hypercube and the interpretive simulator. The portable simulator was constructed to provide simulation on a wider variety of computers. (The interpretive simulator has since been ported to the Sequent and Sun 3 computers.)

Simulation of parallel processors has several important uses. First, simulation provides the computing system designer with a test bed to evaluate inexpensively various parallel architectures or design decisions within a given architecture. Second, users who cannot afford a parallel computing system can develop programs on a simulator. Third, even with a parallel system available, the user may find that the simulator provides more debugging aids and performance information than the actual hardware.

*UNIX is a trademark of AT&T.

The remainder of the report describes how to build hypercube programs with the portable simulator. Section 2 is a guide to the use of the simulator with examples and sample sessions for both C and FORTRAN. Section 3 summarizes the implementation of the simulator.

2. User's Guide

The simulator consists of a controlling process, *mpsim*, and one or more application processes. A hypercube application typically consists of a host process and one or more hypercube node processes. The reader is referred to [4] for a description of constructing hypercube applications using the Intel hypercube. An application program, in C or FORTRAN, is linked with a simulator library, *simlib.a*, which provides the hypercube subroutines and the interface routines to *mpsim*. Commands to *mpsim* control the loading and execution of processes, host and node programs, and enable event tracing. The commands are modeled after the commands used to control an Intel hypercube from the host processor and the syntax used by the Intel simulator [3]. The commands and library routines are described in the following two sections, then sample programs and terminal sessions are presented.

2.1. Command interface.

A hypercube application consists of a host program and one or more node programs. For the simulator, the host and node programs are linked with *simlib.a* and then executed under the control of the program *mpsim*. There are six commands available under *mpsim*:

c [logfile]
h file
l [-n node] dim file
s
t [on]
q

The host program is identified and loaded with the *h* command. There can be only one host process, and it will be the program identified by the last *h* command. The node programs are loaded with the *l* command. The dimension of the hypercube, *dim*, to be simulated must be specified with the name of the node program. The maximum number of subprocesses that a particular UNIX implementation allows determines the maximum dimension. Typically, the maximum dimension is three or four, providing ten or eighteen processes (including the host program and *mpsim*). Optionally, you may explicitly specify the node number, *-n node*, that a node program is to occupy. If the option is not used, then 2^{dim} node processes are loaded, and the same program will be running on each node. Explicitly specifying the node number permits different programs to run on different nodes.

The file on which application-generated messages (*syslog()*) are saved and the simulator events logged is specified with the *c* command. If the file name specified is *stdout* then output will be directed to the controlling terminal. The tracing of simulator events, message send's and receive's, is enabled with *t on*. The default trace file is *SIMLOG*. If the trace file already exists, simulator logging will be appended. After the host and node programs have been specified, simulation is started with the *s* command.

2.2. Simulator subroutines

The message-passing conventions are modeled on those used on the Intel hypercube. Appendix A summarizes these subroutines and supplemental information is available from [4]. The subroutines are described using the syntax of C; corresponding subroutines exist for FORTRAN. Programs written for the simulator should port to the Intel hypercube with no source code changes. To send or receive messages the program must first establish one or more communication channels with *copen*. The argument to *copen* is an arbitrary integer, sometimes called the process id*. *Copen* returns an integer value which can be regarded as a file descriptor in C or a FORTRAN logical unit number. The syntax used to send a message is

sendw(ci, type, mesg, size, node, pid)

where *ci* is the channel identifier established with the *copen*. *Type* is an arbitrary integer which can be used by the application to indicate different kinds of messages. *Mesg* is the address of *size* bytes that will be sent to the task with id *node* on the channel that *node* has opened with the process id of *pid*. Nodes are numbered from 0 to 2^{dim}-1, and the host has an id of 32768. There is a corresponding subroutine *sendmsg* with the same arguments that the Intel host is required to use.

The syntax for receiving a message (or waiting for a message to arrive) is

recvw(ci, type, mesg, size, &length, &node, &pid)

When a message of the specified *type* arrives it is stored starting at *mesg* up to a maximum of *size* bytes. *Length* is set to the actual number of bytes sent, and *node* and *pid* are set to the node number and process id of the sending process. The corresponding host version is

recvmsg(ci, &type, mesg, size, &length, &node, &pid)

Note that the host version does not distinguish on *type;* but rather a message of any *type* is received, and the value of *type* is set. Both *recvw* and *recvmsg* are synchronous (blocking); that is, the program is suspended until a message arrives.

Asynchronous versions of *sendw* and *recvw*, called *send* and *recv*, are supported on the nodes for more complex message management. The *status* function is provided to indicate whether the transmission on the given channel (the argument to *status*) has been completed. Completion of *send* does not mean the message has been received, only that the *mesg* area has been copied into the operating system area. Completion of a *recv* indicates that a message has arrived and that the arguments to *recv* have been set. Testing for the completion of a *recv* must be done with a "busy wait." On the Intel hypercube, the recommended sequence uses *flick()*. *Flick* permits other processes on a node to run, but on the simulator the function has no effect because the simulator supports only one process per node. A code fragment illustrating a proper "busy wait" follows:

```
recv(ci, type, mesg, size, &lth, &node, &pid);
while(status(ci)) flick();
```

* On the Intel cube, it is possible to have more than one task running on a node processor. The process id permits the sending task to select a particular process on the node.

By opening several channels, one can have multiple receives outstanding. The *probe(ci, type)* permits the program to see if a message of a given type is in the receive queue. *Probe* returns the length of the message if one is queued; otherwise -1 is returned. A *recv* must still be issued to retrieve the message.

The host and node programs can write intermediate results or debugging information into the trace file supported by the simulator using *syslog(pid,message).* The character string *message* will be appended to the trace file along with value of the integer *pid.*

Figure 1 illustrates the use of some of the simulator routines in a contrived example. The program,written in C, calculates the product of a matrix (*matrix*) and a vector (*vector*) and prints the resulting vector (*result*). The result is calculated in parallel by creating processes to perform the vector products. The host process (*iph.c*) issues a *copen* then sends to each process a message of type 1 containing a row of the matrix and a message of type 2 containing the vector. Then it waits for each process to send back the resulting inner product.

```
/* iph.c  vector matrix(transpose) inner product using messages */
#include <stdio.h>
#define DIM 4
int vector[DIM] = {2,3,1,4};
int matrix[DIM][DIM] = { 1,2,3,4, 2,3,1,0, 3,3,1,2, 4,3,2,1 };
int result[DIM];
main()  /*  host task */
{
        int d,i,val, type,lth,node,pid;

        d = copen(15);
        for (i=0;i<DIM;i++){            /* start and send data to each node */
                sendmsg(d,1,matrix[i],sizeof(matrix[i]),i,15);
                sendmsg(d,2,vector,sizeof(vector),i,15);
        }
        i=DIM;
        while(i--){   /* wait for results */
                recvmsg(d,&type,&val,sizeof(int),&lth,&node,&pid);
                result[node] = val;
        }

        syslog(3,"a host message");
        for(i=0;i<DIM;i++) printf(" %d",result[i]);
        printf(" (should be 27 14 24 23)");
}

/* ipn.c  node */
#define DIM 4
main()
{
        /* do inner product of two vectors */
        int v1[DIM], v2[DIM], i, sum, d, me, node,pid,lth;

        me=mynode();
        d = copen(15);
        recvw(d,1,v1,sizeof(v1),&lth,&node,&pid);
        recvw(d,2,v2,sizeof(v2),&lth,&node,&pid);
        sum=0;
        for(i=0;i<(lth/ sizeof(int));i++) sum += v1[i] * v2[i];
        sendw(d,3,&sum,sizeof(int),node,pid);
}
```

Figure 1. *C host and node programs for matrix-vector product.*

The node process *ipn.c* issues a *copen* and awaits a message of type 1; then it awaits a message of type 2 containing vectors. The inner product of the two vectors is calculated, and the result sent back to the host program. Additional sample programs are included with the simulator distribution.

2.3. Trace file and post-processors

The simulator can provide extensive debugging and performance information if tracing is enabled with the *t on* command to *mpsim*. If the trace file exists, the trace information will be appended; otherwise a new file is created. Thus it is usually necessary to remove the old trace file between successive runs of an application. The *c filename* command may be used to specify a trace file. If the file name is *stdout* then the trace output is directed to *stdout* and thus may be viewed directly on the terminal as the program runs.

One line is written to the trace file for each simulator event, such as process initiation, process termination, sending a message, or message arrival. Figure 2 is an excerpt from a trace file. Each entry is time-stamped. The trace file entry for a *send* or *recv* includes the node id (*node*) of the originator along with message type, address, and size and destination (or sender) id. The programmer may include his own data in the trace file with *syslog*, which writes an integer (*pid*) and a character string to the trace file. The *cnt* entries indicate the number of processors active at the given time. The active and waiting processors can be deduced from the "waking" and "blocking" substrings of a trace entry. In practice, the trace file can grow quite rapidly, so discretion is advised.

```
recvw clock 210 node 1 pid 15 type 1 lth 16 blocking 1
cnt  3     clock210
recvw clock 230 node 2 pid 15 type 1 lth 16 blocking 2
cnt  2     clock230
recvw clock 250 node 0 pid 15 type 1 lth 16 blocking 0
cnt  1     clock250
send clock 270 node 32768 fpid 15 to 0 pid 15  type 1 lth 16 waking 0
cnt  2     clock270
send clock 300 node 32768 fpid 15 to 0 pid 15  type 2 lth 16
recvw clock 320 node 0 pid 15 type 2 lth 16 from 32768
send clock 350 node 32768 fpid 15 to 1 pid 15  type 1 lth 16 waking 1
cnt  3     clock350
send clock 370 node 0 fpid 15 to 32768 pid 15  type 3 lth 4
recvw clock 660 node 32768 pid 15 type -1 lth 4 from 1
texit node 3 clock 690
cnt  1     clock690
recvw clock 720 node 32768 pid 15 type -1 lth 4 from 3
syslog clock 750 node 32768 id 3 msg a host message
```

Figure 2. *Trace file excerpt.*

The raw trace file can be a very useful debugging aid (see §2.6), but trace files are usually interpreted by post-processors to give performance summaries. Tabular summaries of sends and receives and processor utilization can be displayed with the *nstats* command. A sample output of *nstats* is part of the sample session in Figure 5. Two post-processors, *ccplot* and *trace1*, produce graphical output suitable for use by the UNIX *graph* command. For example,

ccplot tracefile | graph -b | plot -T4010

would plot processor utilization over time on a Tektronix 4010 graphics terminal. *Ccplot* and *trace1* provide more meaningful data when run with the interpretive simulator [1].

2.4. Sample session

Figure 3 is a transcript of a terminal session illustrating how one builds simulator programs in C and invokes post-processors. The files used are part of the simulator distribution tape. The actual location of these files is determined by where the simulator was installed at a given site. The files *iph.c* and *ipn.c* are the vector-matrix programs described in §2.2. The script *sbld* is used to compile the host and node programs and link them with the simulator library *simlib.a*. Typical usage is *sbld file* where *.c* is assumed as the extension to *file*. The executables *iph* and *ipn* produced by *sbld* are run by *mpsim* and the resulting vector is printed. The trace file is analyzed by *nstats*, and a summary of processor utilization and message counts is reported.

```
% sbld iph
% sbld ipn
% mpsim
mpsim> t on
 trace and logging on to SIMLOG
mpsim> h iph
mpsim> l 2 ipn
 dimension 2 cube
mpsim> s
 27 14 24 23 (should be 27 14 24 23)
 mpsim exiting
% nstats SIMLOG
node    start     end    duration   busy    utiliz   sends   recvs
Host      150     800       650      650      100%       8       4
  0       120     410       290      270       93%       1       2
  1        50     500       450      310       69%       1       2
  2        20     590       570      320       56%       1       2
  3        90     690       600      300       50%       1       2

Nodal utilization  67%  Nodal+host utilization  74% sends 12  recvs 12
Gross utilization  47%

Total messages      12      144 bytes
lth   count             bytes
  8       4  33%         16  11%
 16       0   0%          0   0%
 32       8  67%        128  89%
 64       0   0%          0   0%
128       0   0%          0   0%
256       0   0%          0   0%
512       0   0%          0   0%
1024      0   0%          0   0%
2048      0   0%          0   0%
4096      0   0%          0   0%
8192      0   0%          0   0%
16000     0   0%          0   0%

hops   count             bytes
 -1    12 100%         144 100%
  0     0   0%          0   0%
  1     0   0%          0   0%
  2     0   0%          0   0%
  3     0   0%          0   0%
  4     0   0%          0   0%
  5     0   0%          0   0%
  6     0   0%          0   0%
```

Figure 3. *Sample simulator session for C.*

2.5. FORTRAN interface

The simulator subroutines are available to the FORTRAN programmer as well. Figure 4 illustrates some of the simulator FORTRAN subroutines using the sample described in §2.2. The program calculates the inner product of a matrix (*matrix*) with a vector (*vector*) and prints the resulting vector (*result*). The result is calculated in parallel by creating processes to perform the vector products. The host process (*iphf.f*) issues a *copen* and sends a message of type 1 containing a column of the matrix and a message of type 2 containing the vector to each process. Then it waits for each process to send back the resulting inner product.

```
c iphf.f  host
      implicit integer (a-z)
c do simple inner product with snd rcv
      integer matrix(4,4),vector(4), result(4)
      data matrix /1,2,3,4,2,3,1,0,3,3,1,2,4,3,2,1/
      data vector /2,3,1,4/

      d = copen(15)
      do 10 i=1,4
            call sendmsg(d,1,matrix(1,i),16,i-1,15)
            call sendmsg(d,2,vector,16,i-1,15)
10    continue
      do 20 i=1,4
            call recvmsg(d,itype,val,4,lth,node,pid)
            result(node+1) = val
20    continue
      write(*,*)result
      end


c ipnf.f  node
      implicit integer (a-z)
c multiply two vectors
      integer v1(4), v2(4)

      d = copen(15)
      call recvw(d,1,v1,16,lth,node,pid)
      call recvw(d,2,v2,16,lth,node,pid)
      sum = 0
      do 10 i=1,4
10          sum = sum + v1(i) * v2(i)
      call sendw(d,3,sum,4,node,pid)
      end
```

Figure 4. *FORTRAN program for matrix-vector product.*

The node process *ipnf.f* issues a *copen* and awaits a message of type 1 containing a column of the matrix, then it awaits a message of type 2 containing a vector. The inner product of these two vectors is calculated and the result sent back to the host program. Several simulator functions return values of type INTEGER but do not follow the FORTRAN default naming convention, so one must remember to declare these functions (*copen*, *status*, *probe*, *cubedim*, *clock*) as INTEGER.

Figure 5 is a transcript of a terminal session illustrating how one builds FORTRAN simulator programs and invokes post-processors. The files used are part of the simulator distribution tape. The file *ipfh.f* is the matrix-vector host program described above. The script *sfbld* is used to compile the host and node programs and link them to the simulator library *simlib.a*, using a command like *sfbld file*, where *.f* is assumed as the extension to

file. The executables *iphf* and *ipnf* produced by *sfbld* are run by *mpsim* and the resulting vector is printed. The trace file is analyzed by *nstats* and a summary of processor utilization and message counts is reported.

```
% sfbld iphf
% sfbld ipnf
% mpsim
mpsim> t on
 trace and logging on to SIMLOG
mpsim> h iphf
mpsim> l 2 ipnf
 dimension 2 cube
mpsim> s
 27  14  24  23
 mpsim exiting
% nstats SIMLOG
node    start     end    duration   busy    utiliz   sends  recvs
Host     20       780      760      760     100%      8      4
  0     130       410      280      280     100%      1      2
  1     160       500      340      270      79%      1      2
  2      90       590      500      290      58%      1      2
  3      60       690      630      310      49%      1      2

Nodal utilization 72%  Nodal+host utilization 77% sends 12  recvs 12
Gross utilization 50%

Total messages     12      144 bytes
lth   count            bytes
  8      4  33%         16  11%
 16      0   0%          0   0%
 32      8  67%        128  89%
 64      0   0%          0   0%
128      0   0%          0   0%
256      0   0%          0   0%
512      0   0%          0   0%
1024     0   0%          0   0%
2048     0   0%          0   0%
4096     0   0%          0   0%
8192     0   0%          0   0%
16000    0   0%          0   0%

hops   count            bytes
 -1    12 100%        144 100%
  0     0   0%          0   0%
  1     0   0%          0   0%
  2     0   0%          0   0%
  3     0   0%          0   0%
  4     0   0%          0   0%
  5     0   0%          0   0%
  6     0   0%          0   0%
```

Figure 5. *Sample FORTRAN session.*

2.6. Debugging

The simulator provides a number of aids for discovering bugs in a parallel application. To reduce the number of initial bugs, the initial implementation should be kept simple, deferring optimizations for speed or storage savings until later. The program should be written so that it can run with an arbitrary number of nodes and then tested with just

a few. This will reduce the size of the trace file, as well as any debugging output. If possible, the message-passing logic of the application should be isolated and tested.

The trace file provides a wealth of information when things go wrong. Often synchronization problems arise from messages arriving in an unanticipated order. The trace file shows who is sending what to whom and when. Synchronization problems can be reduced by using different "type" fields in the *send* and *recv* calls. Distinct message types also make following the program sequence in the trace file easier. Use of *syslog* to note different phases of the application also aids in reading the trace file. Messages are sometimes not received because the process-id in the *send* does not match the process-id used by the receiving task in its *copen*. Message sizes in the trace file should also be checked. Sometimes a program is changed to send different data, but the user fails to adjust the message size in the *send* or *recv*. Failure to specify proper lengths in *recv* can have fatal results, since other variables in the program may be over-written. In C, care must be taken in specifying addresses of variables, using "*&*" where required.

2.7. Intel iPSC and the Simulator

Our initial algorithm design and program development for message-passing architectures was done on the simulator. We then acquired an Intel iPSC d6 hypercube, and the simulator was modified to utilize calling sequences of the Intel cube [4]. Even with access to a real message-passing machine, the simulator still is a useful tool in program development and algorithm analysis for the following reasons. First, as of this writing, the Intel cube is a single user system (though other users may be doing program development on the cube host while the cube is in use), whereas the simulator permits many users to work concurrently on their own "cubes." Second, the simulator is presently better instrumented than the actual cube for providing debugging information (output directly from the node processes) and performance data (trace files, plots, processor utilization). Finally, the simulator as implemented on the Sequent multiprocessor actually runs faster than the real cube for many programs when using the same number of nodes as Sequent processors (Table 1). Figure 6 compares the message-passing performance of the Intel hypercube to the simulator run on various computers. The figure shows the data rate for sending a message between two adjacent nodes over a range of message sizes.

Developing a program to run correctly on both the simulator and the Intel cube does require some care. The Intel cube is based on the Intel 80286 CPU. If the simulator machine has a different architecture then the source code may have to be modified when moving from the simulator to the actual cube. For example, the default word size for a VAX or Sun is four bytes, but it is only two bytes for the Intel cube and its host. Thus the default precision for *int* in C and INTEGER in FORTRAN is different for the two machines. For arguments to simulator subroutines, one may just declare the arguments as *int* or INTEGER and the calls should be compatible between the simulator (which uses four-byte arguments) and the Intel cube (which uses two bytes). However, other areas of the user's code may give incorrect results because of the different word size.

The simulator is not a true simulation of the Intel cube for a number of reasons. There may be only one process per node on the simulator. The message-passing delays do not model the behavior of a hypercube network. The simulator does not account for message delays induced by traffic from other nodes nor does it account for compute delays induced by message routing. Nevertheless, the simulator provides a useful and reasonably accurate predictor of actual cube performance and is a powerful tool for performance analysis and debugging.

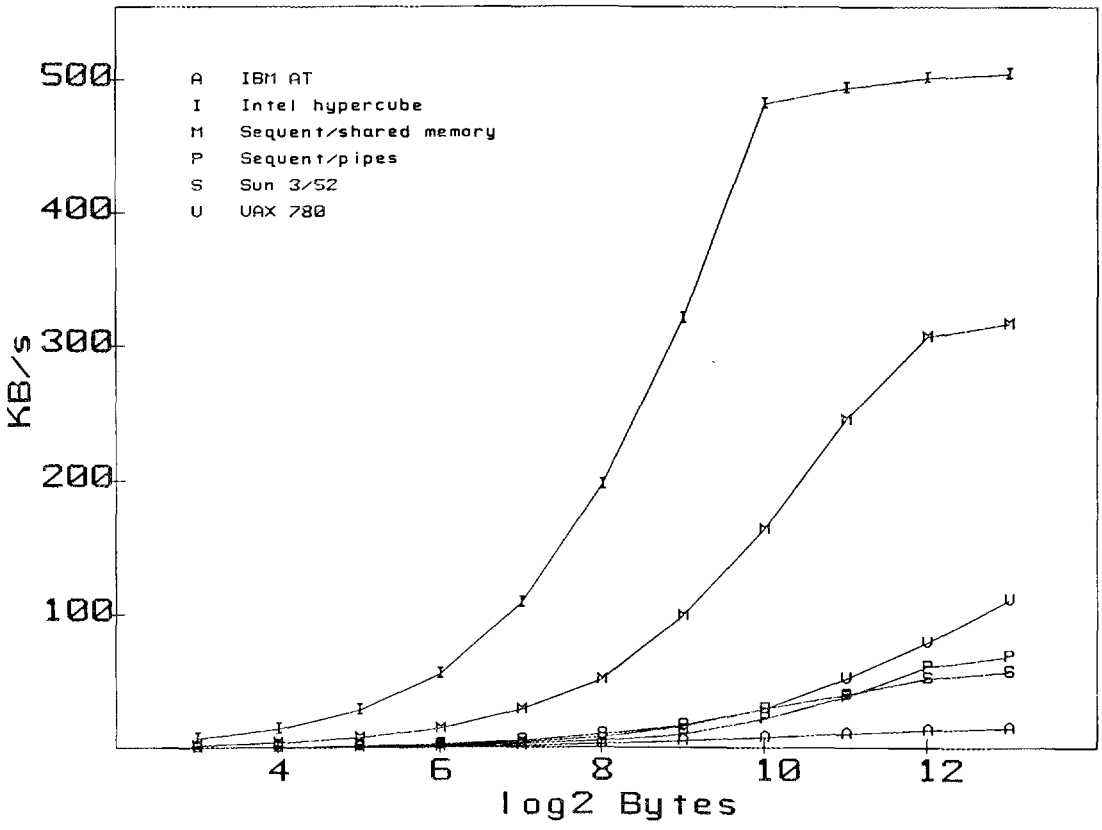

Figure 6. Simulator and hypercube message-passing performance.

3. Implementation.

3.1. Portable implementation.

The simulator consists of a controlling program, *mpsim*, and a run-time library for the user's application processes, host and node programs. The library and *mpsim* are written in C, and the library supports both C and FORTRAN. The implementation uses standard UNIX system calls, specifically using *fork()* and *signal()* and pipes for inter-process communication. All hypercube message passing among the application processes is handled by *mpsim*. The application processes do not communicate directly with each other. Three pipes are used by *mpsim* to communicate with the application processes. The request-pipe is written by all application processes and read by *mpsim*. The request message is just the integer UNIX process id of the requesting process. It is assumed that this write is an atomic operation, a necessity, since there could be multiple simultaneous write's. (An atomic write could be assured by just writing the low order byte of the process id.) A message-pipe is read and written by both an application process and *mpsim*, with access synchronized through UNIX signals. Finally, an acknowledgement-pipe is written by the application process to let *mpsim* know that the message-pipe has been emptied.

The user identifies the application processes to *mpsim* and starts the simulation. *Mpsim* creates file descriptors for the pipes and forks the application processes, which inherit the file descriptors. *Mpsim* then reads the request-pipe waiting for a service request from one of the application processes. When a request is received, a signal is issued to the

requesting process permitting it to write its request in the message-pipe. *Mpsim* reads the message-pipe and performs the desired service. If a reply is expected, *mpsim* writes the reply into the message-pipe and again signals the application process. *Mpsim* then reads the acknowledgement-pipe waiting for the acknowledgement-write from the application and then returns to reading the request-pipe. *Mpsim* also detects when a process exits, and when all processes have exited, *mpsim* will terminate. *Mpsim* maintains a queue of processes awaiting messages and a queue of messages to be delivered and manages the trace file.

The application processes are linked with the simulator library, *simlib.a*. The library includes FORTRAN interface routines to translate FORTRAN argument passing to C conventions. The application processes are run as sub-processes of *mpsim* and inherit the pipe file descriptors. Requests to use the message-pipe are made by writing on the request-pipe and then issuing a *sigpause()* to suspend the process until *mpsim* issues the signal to proceed. A service-request message and possibly user data (for a *send*) are then written on the message-pipe. If data is expected in reply, then the application process again pauses until *mpsim* signals that the requested data is available. The library routines maintain a channel data structure that is used by the hypercube subroutines and is initialized when the first request to *mpsim* is issued.

3.2. Sequent implementation.

On shared-memory multiprocessors such as a Sequent or Encore, the simulator processes can execute concurrently on the multiple processors yielding very high simulator performance. To increase the performance even further, *mpsim* and the application library were modified to use the shared-memory structures of the Sequent in place of the UNIX pipes. (The Sequent version, *smpsim*, is distributed separately.) A shared memory region with an associated lock is created by *smpsim* and mapped by the library to the application processes. An application process acquires the region, copies in its hypercube message or request, and signals *smpsim* by clearing a spin-lock in the shared region. *Smpsim* copies the message into its message queues, and then notifies the application by means of another spin-lock that the request has been completed. The application then releases the memory lock. Using shared memory instead of pipes increases the performance of message passing by an order of magnitude over pipes (Figure 6). Figure 6 also compares the speed of simulator message passing to that of a real hypercube.

Cholesky factorization time (seconds)	
Intel hypercube	9
smpsim (Sequent)	9
mpsim (Sequent)	54
ppsim (Sequent)	25
ppsim-aspp (Sequent)	2354
mpsim (Vax)	120
ppsim (Vax)	17
ppsim-aspp (Vax)	1627
mpsim (Sun)	44
ppsim (Sun)	18
ppsim-aspp (Sun)	2730
mpsim (Intel 310)	157

Table 1. *Hypercube and simulator performance.*

Table 1 compares the relative compuational and message-passing performance of the various simulators with that of the a real hypercube for a Cholesky matrix factorization. A matrix of size *128x128* is factored on four nodes and timed on an Intel hypercube and on several machines using various simulators. *Smpsim* is the shared-memory version of *mpsim* and performs the factorization on four processors of the Sequent in approximately the same time as the Intel hypercube. *Ppsim* is an interpretive simulator [1] that provides instruction-level trace information in *aspp*-mode.

References

[1] T. H. Dunigan, *A Message-passing Multiprocessor Simulator*, Tech. Rept. ORNL/TM-9966, Oak Ridge National Laboratory, Oak Ridge, TN (1986).

[2] G. C. Fox and S. W. Otto, *Algorithms for concurrent processors*, Physics Today, May 1984, pp. 50-59.

[3] Intel, *iPSC Hypercube Simulator*, Intel 310104-003, Portland, Oregon, October, 1986.

[4] Intel, *iPSC User's Guide*, Intel 17455-03, Portland, Oregon, October, 1985.

[5] C. L. Seitz, *The cosmic cube*, Comm. ACM, 28 (1985), pp. 22-33.

Appendix A

Simulator Manual Pages

NAME

mpsim - portable message-passing multiprocessor simulator

SYNOPSIS

mpsim commands:

c [filename]	disable logging or log to *filename*
h host	load host with file *host*
l [-n nodeno dim node	load *nodeno* node or all nodes of a cube of dimension *dim* with file *node*
q	quit
s	start simulation
t [on]	enable or disable trace

The host and node programs should be linked with the library *simlib.a* which contains the following subroutines:

```
int copen(int pid);
void cclose();

void send(int d, int type, char *msg, int msglth,
        int dstnode, int dstpid);
void sendw(int d, int type, char *msg, int msglth,
        int dstnode, int dstpid);
void sendmsg(int d, int type, char *msg, int msglth,
        int dstnode, int dstpid);

void recv(int d, int type, char *msg, int maxlth,
        int *msglth, int *srcnode, int *srcpid);
void recvw(int d, int type, char *msg, int maxlth,
        int *msglth, int *srcnode, int *srcpid);
void recvmsg(int d, int *type, char *msg, int maxlth,
        int *msglth, int *srcnode, int *srcpid);

int probe(int d, type);
int status(int d);

int mynode();
int cubedim();
int clock();
void syslog(int pid, char *msg);
void flick();
```

DESCRIPTION

A simulator driver task, *mpsim*, provides a set of commands for loading and controlling a set of hypercube application programs built with the simulator library *simlib.a*. The simulator can be used to execute programs developed for the Intel hypercube, and a trace file is provided to assist in debugging or performance analysis. The simulator uses the UNIX fork and pipe facilities and has been tested on a Intel 310 with XENIX, an IBM PC/AT with XENIX, a DEC VAX with UNIX 4.2, and a Sequent Balance 8000 with DYNIX. Since the simulation will spawn multiple processes, you may wish to use *nice(1)* to reduce system load. Note that simulation on the Sequent will utilize multiple processors. The simulation does not provide the detailed timings, large number of processors, or message-

passing model of the simulator described in *intel(l)*, but does permit porting of programs between the Intel cube and the simulator with no source code changes. In addition, there are no restrictions on the use of COMMON or global variables in C. The programmer compiles and links his host and node programs with *simlib.a* and then runs *mpsim* to load the application programs into the simulator cube and start the simulation.

The *c* commmand (cubelog) turns off logging with no arguments, or if a file name is provided, enables logging to the given file. Information will be appended to the file. The *h* command (host) specifies an executable file that will be the application host program. The *l* command (load — you may substitute *m* for *l)* loads the simulator nodes with the program *node*. A specific node may be loaded with the *-n* option. *Dim* specifies the dimension of the cube and must be in the range 0-4. (On some XENIX systems the maximum dimension may be 3.) The *t* command (trace) enables or disables tracing of simulator events *send*'s and *recv*'s. Once all application programs are "loaded" the *s* command (start) will start the simulation, you can exit with the *q* command (quit). If all processes exit then the simulator will exit, otherwise it will be necessary to use CTRL-C to terminate the simulation. A sample session might be

```
cc -o host host.c simlib.a -lm
cc -o node node.c simlib.a -lm
mpsim
  t on
  l 3 node
  h host
  s
```

The functions provided in *simlib.a* mirror those described in the Intel *iPSC User's Guide*. The programmer must first establish one or more cube communication channel data structures with calls to *copen*. *copen* takes a value to be used as process identifier and returns a descriptor, of type int, that is used in subsequent message-passing functions. A process is addressed by its node number and process identifier. *cclose* frees the channel data structure.

send and *sendw* send the message pointed to by msg to the process at node dstnode with process id dstpid. The type and size of the message (msglth) in bytes are also provided. *send* returns immediately, but one cannot use the message area until *status* returns FREE (0), indicating that the kernel has sent the message. *sendw* does not return until the message has been sent. (This does not imply that the message has been received.) *sendmsg* behaves exactly like *sendw* but is intended as the host version for compatibilty with INTEL cube.

recv and *recvw* await the arrival of a message of the given type for the node and process id associated with int d. The functions provide addresses to store the message, the actual length of the message, and the node and process id of the sender. For *recvw* the process blocks until a message of the given type arrives. For *recv* the process may continue processing after issuing the *recv*, and when *status* returns a value of FREE (0), then a message of the given type has arrived. Upon receipt of the message, the simulator sets the srcnode and srcpid to those of the sender, sets the msglth to the length of the received message, and copies the message into msg. No more than maxlth bytes are copied. Messages are handled in a FIFO fashion. *recvmsg* behaves like *recvw* except it does not discriminate on message type, rather the type of the message is returned along with the message in accordance with the INTEL cube. *sendmsg* and *recvmsg* are intended (by INTEL) to be used only by the host processes, but the simulator permits node usage as well.

probe determines if a message of the given type is available for the node and process id associated with the given int. If a message is available, *probe* returns the length of the message; otherwise, a value of -1 is returned. One must issue a *recv* actually to fetch the

message. Note, the channel data structure should not be in use by other message-passing functions. *status* returns a value of BUSY (1) or FREE (0) indicating whether the given int data structure is in use or not. For *send*, BUSY indicates that the kernel has not yet sent the message. For *recv*, BUSY indicates that the desired message has not arrived.

mynode returns the node number of the process. The host has a node number of x8000 (32768). *cubedim* returns the dimension of the cube. *clock* returns the present value of the time-of-day clock in milliseconds. *syslog* places the given message and pid in the trace file. *flick* relinquishes control from the given process to other runnable processes. *flick* is usually used in busy-wait conditions with *status* following a *recv* or with *probe*.

POST PROCESSORS

If tracing has been enabled then a trace file is produced with simulator data that can be summarized by *nstats* or *ccplot*. *nstats tracefile* will produce a per-node summary of compute time and sends and receives. *ccplot tracefile > plotfile* will produce a plotfile that can be plotted with various plotting programs such as *graph(1)*.

FORTRAN

The message passing subroutines may also be called from *f77* programs. At present, XENIX Intel FORTRAN is not supported.

FILES

The following files are provided; the actual location is site dependent.

simlib.a	simulator subroutines
mpsim	simulator driver
sbld	sample C build script
sfbld	sample f77 build script

SEE ALSO

ppsim(1), hep(1), intel(1) and Intel's *iPSC User's Guide*

BUGS

The delay in message passing is due to delays in UNIX pipes and does not reflect a hypercube structure. Messages are passed through the simulator driving task *mpsim*. The clock used is just the computer's time of day clock and has a resolution of only 20 milliseconds. Running multiple processes on a single node is not supported. The *handler* function and the Intel dynamic loader functions are not presently implemented.

AUTHOR

T. H. Dunigan

NAME

smpsim - Sequent message-passing multiprocessor simulator

SYNOPSIS

smpsim commands:

c [filename]	disable logging or log to *filename*
h host	load host with file *host*
l [-n nodeno] dim node	load *nodeno* node or all nodes of a cube of dimension *dim* with file *node*
q	quit
s	start simulation
t [on]	enable or disable trace

The host and node programs should be linked with the library *ssimlib.a* which contains the following subroutines:

```
int copen(int pid);
void cclose();

void send(int d, int type, char *msg, int msglth,
        int dstnode, int dstpid);
void sendw(int d, int type, char *msg, int msglth,
        int dstnode, int dstpid);
void sendmsg(int d, int type, char *msg, int msglth,
        int dstnode, int dstpid);

void recv(int d, int type, char *msg, int maxlth,
        int *msglth, int *srcnode, int *srcpid);
void recvw(int d, int type, char *msg, int maxlth,
        int *msglth, int *srcnode, int *srcpid);
void recvmsg(int d, int *type, char *msg, int maxlth,
        int *msglth, int *srcnode, int *srcpid);

int probe(int d, type);
int status(int d);

int mynode();
int cubedim();
int clock();
void syslog(int pid, char *msg);
void flick();
```

DESCRIPTION

A simulator driver task, *smpsim*, provides a set of commands for loading and controlling a set of hypercube application programs built with the simulator library *ssimlib.a*. The simulator can be used to execute programs developed for the Intel hypercube, and a trace file is provided to assist in debugging or performance analysis. The simulator uses the UNIX fork and Seuqent shared memory facilities. The simulation does not provide the detailed timings, large number of processors, or message-passing model of the simulator described in *intel(l)*, but does permit porting of programs between the Intel cube and the simulator with no source code changes. In addition, there are no restrictions on the use of COMMON or global variables in C. The programmer compiles and links his host and node

programs with *ssimlib.a* and then runs *smpsim* to load the application programs into the simulator cube and start the simulation.

The *c* commmand (cubelog) turns off logging with no arguments, or if a file name is provided, enables logging to the given file. Information will be appended to the file. The *h* command (host) specifies an executable file that will be the application host program. The *l* command (load — you may substitute *m* for *l)* loads the simulator nodes with the program *node*. A specific node may be loaded with the *-n* option. *Dim* specifies the dimension of the cube and must be in the range 0-4. The *t* command (trace) enables or disables tracing of simulator events *send*'s and *recv*'s. Once all application programs are "loaded" the *s* command (start) will start the simulation, you can exit with the *q* command (quit). If all processes exit then the simulator will exit, otherwise it will be necessary to use CTRL-C to terminate the simulation. A sample session might be

```
cc -O -o host host.c ssimlib.a -lm -lpp
cc -O -o node node.c ssimlib.a -lm -lpp
smpsim
  t on
  l 3 node
  h host
  s
```

The functions provided in *ssimlib.a* mirror those described in the Intel *iPSC User's Guide*. The programmer must first establish one or more cube communication channel data structures with calls to *copen*. *copen* takes a value to be used as process identifier and returns a descriptor, of type int, that is used in subsequent message-passing functions. A process is addressed by its node number and process identifier. *cclose* frees the channel data structure.

send and *sendw* send the message pointed to by msg to the process at node dstnode with process id dstpid. The type and size of the message (msglth) in bytes are also provided. *send* returns immediately, but one cannot use the message area until *status* returns FREE (0), indicating that the kernel has sent the message. *sendw* does not return until the message has been sent. (This does not imply that the message has been received.) *sendmsg* behaves exactly like *sendw* but is intended as the host version for compatibilty with INTEL cube.

recv and *recvw* await the arrival of a message of the given type for the node and process id associated with int d. The functions provide addresses to store the message, the actual length of the message, and the node and process id of the sender. For *recvw* the process blocks until a message of the given type arrives. For *recv* the process may continue processing after issuing the *recv*, and when *status* returns a value of FREE (0), then a message of the given type has arrived. Upon receipt of the message, the simulator sets the srcnode and srcpid to those of the sender, sets the msglth to the length of the received message, and copies the message into msg. No more than maxlth bytes are copied. Messages are handled in a FIFO fashion. *recvmsg* behaves like *recvw* except it does not discriminate on message type, rather the type of the message is returned along with the message in accordance with the INTEL cube. *sendmsg* and *recvmsg* are intended (by INTEL) to be used only by the host processes, but the simulator permits node usage as well.

probe determines if a message of the given type is available for the node and process id associated with the given int. If a message is available, *probe* returns the length of the message; otherwise, a value of -1 is returned. One must issue a *recv* actually to fetch the message. Note, the channel data structure should not be in use by other message-passing functions. *status* returns a value of BUSY (1) or FREE (0) indicating whether the given int data structure is in use or not. For *send*, BUSY indicates that the kernel has not yet sent the message. For *recv*, BUSY indicates that the desired message has not arrived.

mynode returns the node number of the process. The host has a node number of x8000 (32768). *cubedim* returns the dimension of the cube. *clock* returns the present value of the time-of-day clock in milliseconds. *syslog* places the given message and pid in the trace file. *flick* relinquishes control from the given process to other runnable processes. *flick* is usually used in busy-wait conditions with *status* following a *recv* or with *probe*.

POST PROCESSORS

If tracing has been enabled then a trace file is produced with simulator data that can be summarized by *nstats* or *ccplot*. *nstats tracefile* will produce a per-node summary of compute time and sends and receives. *ccplot tracefile > plotfile* will produce a plotfile that can be plotted with various plotting programs such as *graph(1)*.

FORTRAN

The message passing subroutines may also be called from FORTRAN programs.

FILES

The following files are provided; the actual location is site dependent.

ssimlib.a	simulator subroutines
smpsim	simulator driver
ssbld	sample C build script
ssfbld	sample f77 build script

SEE ALSO

ppsim(1), hep(1), intel(1) and Intel's *iPSC User's Guide*

BUGS

The delay in message passing is due to delays in copying messages between the application area and shared memory and does not reflect a hypercube structure. Messages are passed through the simulator driving task *smpsim*. The clock used is just the computer's time of day clock and has a resolution of only 20 milliseconds. Running multiple processes on a single node is not supported. Running more processes than Sequent processors may be undesirable since busy-waits (spin locks) are used for synchronization. The *handler* function and the Intel dynamic loader functions are not presently implemented.

AUTHOR

T. H. Dunigan

DISTRIBUTION OF ORNL/TM-10410

1. J. Barhen
2-31. T. H. Dunigan
32. Y. H. Etheridge
33-34. R. F. Harbison/ Mathematical Sciences Library
35. G. A. Geist
36. J. A George
37.-41. M. T. Heath
42.-46. J. K. Ingersoll
47. J. M. Jansen
48.-52. F. C. Maienschein
53. E. Ng
54. C. H. Romine
55.-59. R. C. Ward
60. J. Wooten
61. A. Zucker
62. Central Research Library
63. K-25 Plant Library
64. ORNL Patent Office
65. Y-12 Technical Library Document Reference Section
66. Laboratory Records Department--RC
67.-68. Laboratory Records Department
69. P. W. Dickson, Jr. (Consultant)
70. G. H. Golub (Consultant)
71. R. W. Haralick (Consultant)
72. D. Steiner (Consultant)

EXTERNAL DISTRIBUTION

73. Dr. Donald M. Austin, Office of Scientific Computing, Office of Energy Research, ER-7, Germantown Building, U.S. Department of Energy, Washington, DC 20545
74. Dr. Bill L. Buzbee, C-3, Applications Support & Research, Los Alamos National Laboratory, P.O. Box 1663, Los Alamos, NM 87545
75. Dr. John Cavallini, ER-7, GTN, Office of Scientific Computing, Department of Energy, Washington, D.C. 20545
76. Dr. Melvyn Ciment, National Science Foundation, 1800 G Street N.W., Washington, DC 20550
77. Dr. James Corones, Ames Laboratory, Iowa State University, Ames, IA 50011
78. Dr. George J. Davis, Department of Mathematics, Georgia State University, Atlanta, GA 30303
79. Dr. Jack J. Dongarra, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439
80. Prof. Geoffrey C. Fox, Associate Provost for Computing, California Institute of Technology, Pasadena, CA 91125
81. Dr. R. E. Funderlic, Dept. of Computer Science--Box 8206, North Carolina State University, Raleigh, NC 27695-8206
82. Dr. Robert Huddleston, Computation Department, Lawrence Livermore National Laboratory, P.O. Box 808, Livermore, CA 94550
83. Prof. Malvyn Kalos, Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10013
84. Dr. Jeff Kien, Ametek Corp., 610 North Santa Anita Avenue, Arcadia, CA 91006
85. Prof. David J. Kuck, Center for Supercomputing R&O, University of Illinois, 1384 W. Springfield Avenue, Urbana, IL 61801

86. Dr. Joseph Liu, Department of Computer Science, York University, 4700 Keele Street, Downsview, Ontario, Canada M3J 1P3

87. Dr. Paul C. Messina, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439

88. Dr. Cleve Moler, Intel Scientific Computers, 15201 N.W. Greenbrier Parkway, Beaverton, OR 97006

89. Dr. James M. Ortega, Department of Applied Mathematics, University of Virginia, Charlottesville, VA 22903

90. Dr. John F. Palmer, NCUBE Corporation, 915 E. LaVieve Lane, Tempe, AZ 85284

91. Dr. Jesse Poore, Computer Science Dept., 107 Ayres Hall University of Tennessee Knoxville, TN 37996

92. Dr. Garry Rodrigue, Numerical Mathematics Group, Lawrence Livermore Laboratory, Livermore, CA 94550

93. Capt. John P. Thomas, Air Force Office of Scientific Research, Building 410, Bolling Air Force Base, Washington, DC 20332

94. Dr. Robert G. Voigt, ICASE, MS 132-C, NASA Langley Research Center, Hampton, VA 23665

95. Office of Assistant Manager for Energy Research and Development, Department of Energy, Oak Ridge Operations Office, Oak Ridge, TN 37830

96.-125. Technical Information Center.