ornl

ORNL/TM-10685

OAK RIDGE
NATIONAL
LABORATORY

MARTIN MARIETTA

Hypercube Simulation on a Local Area Network

T. H. Dunigan

OAK RIDGE NATIONAL LABORATORY
CENTRAL RESEARCH LIBRARY
CIRCULATION SECTION
4500N ROOM 175
LIBRARY LOAN COPY
DO NOT TRANSFER TO ANOTHER PERSON
If you wish someone else to see this report, send in name with report and the library will arrange a loan.

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

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

ORNL/TM-10685

Engineering Physics and Mathematics Division

Mathematical Sciences Section

HYPERCUBE SIMULATION ON A LOCAL AREA NETWORK

T. H. Dunigan

Date Published - November 1988

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

MARTIN MARIETTA ENERGY SYSTEMS LIBRARIES

3 4456 0379909 4

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 6

  2.4 Sample session 8

  2.5 FORTRAN interface 8

  2.6 Debugging 11

  2.7 Compatibility with the Intel iPSC/1 hypercube 11

3. Implementation 12

4. Performance 12

5. Future work 14

References 16

Appendix A: Simulator Manual Pages 17

Hypercube Simulation on a Local Area Network

T. H. Dunigan

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

ABSTRACT

We describe the structure and use of a distributed computing environment based on a hypercube programming model that runs on most UNIX* systems with TCP/IP network support. The package 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 simulated by a collection of computers on a local area network. Each node of the simulated hypercube is a workstation on the network. The simulator produces a trace file that can be used for debugging, performance analysis, or graphical display.

1. Introduction.

This report describes a distributed computing environment based on a hypercube programming model that will run on most UNIX-TCP/IP computing systems. The package has been run on UNIX 4.x BSD and derivative systems (including DEC's Ultrix, Sequent Dynix, Sun, and IBM RT). The package provides a programming library that can be used to develop distributed applications in either C or FORTRAN. The programmer can use one or more computers on a TCP/IP network to implement a parallel application.

The package is a supplement to our interpretive simulator [1] and is derived from our portable hypercube simulator *mpsim* [2]. The distributed computing environment described in this report provides hypercube simulation and permits multiple computers on a network to work concurrently on an application and provide real speedups from parallel execution. The programming model is that of a message-passing, distributed-memory multiprocessor, based on the Intel iPSC/1 hypercube. Application codes written to run under the package are compatible with the iPSC/1 and could be easily modified to run on other hypercubes or other message-passing multiprocessors.

*UNIX is a trademark of AT&T.

The availability of inexpensive, high-performance computers connected by high-speed networks was the primary motivation for the development of this distributed computing package. The distributed environment permits the hypercube model to be extended to include machines with large virtual memory, graphics, and large file storage. The package provides a means to solve very large, computationally intensive problems using the growing corporate resource of high-performance workstations and other computers connected by high-speed local area networks. As a hypercube simulator, the package permits us to test and develop hypercube applications for the next generation of hypercubes. The trace facility and other debugging aids available with the package can provide valuable debugging and performance information often not available on real hypercubes. For those who cannot afford a hypercube, the package provides an environment to test and develop hypercube applications.

The remainder of the report describes how to build hypercube programs with the distributed computing package. Section 2 is a guide to the use of the package with examples and sample sessions for both C and FORTRAN. Section 3 summarizes the implementation of the package, and section 4 analyzes its performance.

2. User's Guide

The simulator consists of a controlling process, *dcube*, 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 [7] for a description of constructing hypercube applications using the Intel iPSC/1 hypercube and to [4] and [9] for a survey of hypercube applications. An application program, in C or FORTRAN, is linked with a simulator library, *libdcube.a*, which provides the hypercube subroutines and the interface routines to *dcube*. Commands to *dcube* 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 the Intel iPSC/1 hypercube from the host processor and the syntax used by the Intel simulator [6]. 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 *libdcube.a* and then executed under the control of the program *dcube* with assistance from the program *rmtdcube*. There are seven commands available under *dcube*:

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

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. The maximum number of subprocesses that a particular UNIX implementation allows determines the maximum dimension when using one machine to perform the application. Typically, the maximum dimension is three or four, providing ten or eighteen processes (including the host program and *dcube*). Using multiple machines up to 32 processes may be used. Optionally, you may explicitly specify the node number, *-n node*, and machine, *-h remote*, that a node program is to occupy. If the options are not used, then 2^{dim} node processes are loaded on the current machine, and the same program will be running on each node. Explicitly specifying the node number permits different programs to run on different nodes. If a host name is not specified with the *-h* option, then all of the programs will be run on the current machine. It is recommended that the host program run on the current machine to facilitate terminal input-output. See the manual entry in Appendix A for sample command sequences. When using multiple machines, it may be necessary to copy the node programs to the appropriate machines if a network file system is not available. The support program *rmtdcube* must be executable on every machine as well, and the machine must support the *rsh* facility.

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*. After the host and node programs have been specified, simulation is started with the *s* command. One may place a sequence of commands in a command file, and then use the *u* command to execute the command file.

2.2. Simulator subroutines

The message-passing conventions are modeled on those used on the Intel iPSC/1 hypercube. Appendix A summarizes these subroutines and supplemental information is available from [7]. The subroutines are described using the syntax of C; corresponding subroutines exist for FORTRAN. Programs written for the simulator should port to the iPSC/1 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

* 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.

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();
```

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 containing a row of the matrix, then it awaits a message of type 2 containing the vector. 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 *dcube*. 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   clock   210
recvw clock 230 node 2 pid 15 type 1 lth 16 blocking 2
cnt 2   clock   230
recvw clock 250 node 0 pid 15 type 1 lth 16 blocking 0
cnt 1   clock   250
send clock 270 node 32768 fpid 15 to 0 pid 15  type 1 lth 16 waking 0
cnt 2   clock   270
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   clock   350
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   clock   690
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].

```
sun% dbld iph
sun% dbld ipn
sun% cat ip.in
h iph
l -c 2
l -n 0 -h mthsun ipn
l -n 1 -h crfsun ipn
l -n 2 -h yhesun ipn
l -n 3 -h puwsun ipn
sun% dcube
  dcube v1.0 7/23/87
dcube> u ip.in
 dimension 2 cube
dcube> t on
 trace and logging on to SIMLOG
dcube> s
dcube starting remotes  . . . . .started.
 27 14 24 23 (should be 27 14 24 23)
 dcube exiting
sun% nstats SIMLOG
node     start            end   duration    busy   utiliz   sends        recvs
HOST         20        120        100        100   100%        8            4
   0         40        100         60         60   100%        1            2
   1         40         80         40         40   100%        1            2
   2         40        100         60         40    67%        1            2
   3         40        120         80         60    75%        1            2

Nodal utilization  85%  Nodal+host utilization  88% sends 12  recvs 12
Gross utilization  60%

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.*

2.4. 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 *dbld* is used to compile the host and node programs and link them with the simulator library *libdcube.a*. Typical usage is *dbld file* where *.c* is assumed as the extension to *file*. The executables *iph* and *ipn* produced by *dbld* are run by *dcube* using a command file *ip.in* and the resulting vector is printed. It is assumed that the executables have been copied to the appropriate network nodes or are accessible by a network file system. The trace file is analyzed by *nstats*, and a summary of processor utilization and message counts is reported.

```
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.*

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.

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 these functions (*copen, status, probe, cubedim, clock*) must be declared 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 *dfbld* is used to compile the host and node programs and link them to the simulator library *libdcube.a*, using a command like *dfbld file*, where *.f* is assumed as the extension to *file*. The executables *iphf* and *ipnf* produced by *dfbld* are run by *dcube* using the command file *ipf.in* and the resulting vector is printed. It is assumed that the executables have been copied to the appropriate network nodes or are accessible by a network file system. The trace file is analyzed by *nstats* and a summary of processor utilization and message counts is reported.

```
sun% dfbld iphf
iphf.f:
MAIN:
sun% dfbld ipnf
ipnf.f:
 MAIN:
sun% cat ipf.in
h iphf
l -c 2
l -n 0 -h mthsun ipnf
l -n 1 -h crfsun ipnf
l -n 2 -h yhesun ipnf
l -n 3 -h puwsun ipnf
sun% dcube
  dcube v1.0 7/23/87
dcube> u ipf.in
 dimension 2 cube
dcube> t on
 trace and logging on to SIMLOG
dcube> s
dcube starting remotes  . . . . .started.
  27  14  24  23
 dcube exiting
sun% nstats SIMLOG
node  start        end  duration    busy utiliz    sends    recvs
HOST        20       420       400        400   100%      8        4
   0        40       260       220        220   100%      1        2
   1        40       300       260        100    38%      1        2
   2        40       300       260        100    38%      1        2
   3        40       300       260         80    31%      1        2

Nodal utilization  52%  Nodal+host utilization  62% sends 12  recvs 12
Gross utilization  45%

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. As a general rule, the initial implementation of an application 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, initially on one machine. 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. Compatibility with the Intel iPSC/1 hypercube

Dcube provides both an environment for developing distributed applications and a simulation of the Intel iPSC/1 hypercube. The simulation is sufficiently close to the actual iPSC/1 programming environment that, in most cases, no changes to the source code are required. Our initial algorithm design and program development for message-passing architectures were done on simulators. We then acquired an Intel iPSC/1 d6 hypercube, and the simulator was modified to utilize calling sequences of the Intel cube [7]. Even with access to a real message-passing machine, simulators still are useful tools in program development and algorithm analysis for the following reasons. First, the simulator is presently better instrumented than the iPSC/1 hypercube for providing debugging information and performance data (trace files, plots, processor utilization). Second, the simulator as implemented with a collection of workstations runs faster than the real cube for some programs (see section 4).

Developing a program to run correctly on both *dcube* and the Intel iPSC/1 does require some care. The iPSC/1 is based on the Intel 80286 CPU. If the machines used by *dcube* have a different architecture then the source code may have to be modified when moving from the simulator to the iPSC/1 hypercube. For example, the default word size for a VAX or Sun is four bytes, but it is only two bytes for the iPSC/1 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 iPSC/1 (which uses two bytes). However, other areas of the user's code may give incorrect results because of the different word

size.

A distributed collection of machines is not a true simulation of the Intel iPSC/1 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, *dcube* provides a useful and reasonably accurate predictor of actual cube performance and is a powerful tool for performance analysis and debugging.

3. Implementation.

Dcube is based on the portable hypercube simulator, *mpsim* [2]. The distributed computing package consists of a controlling program, *dcube*, a support process, *rmtdcube*, and *libdcube.a*, a run-time library for the user's application processes, host and node programs. The library and control programs are written in C, and the library supports both C and FORTRAN. The implementation uses standard UNIX system calls and TCP/IP socket library routines. All message passing among the application processes is handled by *dcube*. The application processes do not communicate directly with each other.

The user identifies the application processes to *dcube* and starts the distributed computation. *Dcube* uses *rsh* to invoke *rmtdcube* on the participating machines. Each *rmtdcube* opens a TCP/IP socket with *dcube* and then starts an application process with *execl()*. *Dcube* maintains a queue of processes awaiting messages and a queue of messages to be delivered and manages the trace file. *Dcube* exits when all application processes have terminated.

The application processes are linked with the simulator library, *libdcube.a*. The library includes FORTRAN interface routines to translate FORTRAN argument passing to C conventions. The library translates the hypercube subroutine calls to message exchanges with *dcube*. The application processes are spawned from *rmtdcube* and inherit the socket file descriptor for the controlling task *dcube*. Messages are transferred between the application processes through the *dcube* process using the TCP/IP socket. If an application is not ready to receive a message, the message is queued by *dcube* until it is requested. If an application requests a message that has not yet arrived, the application process is blocked on the TCP/IP socket *read()* request to *dcube*. Since application messages must pass through *dcube*, there is some loss of performance, but such an implementation simplifies message queueing, does not require an application to forward messages, and provides a central point for tracing the progress of the parallel application.

4. Performance.

Table 1 compares the relative computational and message-passing performance of our various simulators with that of the iPSC/1 hypercube for a Cholesky matrix factorization. A matrix of size *128×128* is factored on four nodes and timed on an Intel iPSC/1 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 iPSC/1 hypercube. *Ppsim* is an interpretive simulator [1] that provides instruction-level trace information in *aspp*-mode.

Cholesky factorization time (seconds)	
Intel iPSC/1	9
smpsim (Sequent B8000)	9
dcube (5 Sun 3/50)	17
ppsim (Vax 780)	17
ppsim (Sun 3/50)	18
ppsim (Sequent B8000)	25
dcube (Sequent B8000)	36
dcube (5 IBM RTs)	44
mpsim (Sun 3/50)	44
mpsim (Sequent B8000)	54
mpsim (Vax 780)	120
mpsim (Intel 310)	157
ppsim-aspp (Vax 780)	1627
ppsim-aspp (Sequent B8000)	2354
ppsim-aspp (Sun 3/50)	2730

Table 1. *Hypercube and simulator performance.*

Figure 6 compares the message-passing performance of *dcube* and the Intel iPSC/1 hypercube to our various simulators run on various computers. The figure shows the data rate for sending a message between two adjacent nodes over a range of message sizes.

Empirically for most hypercubes, the communication time for a one-hop message is a linear function of the size of the message [3]. That is, the time T to transmit a one-hop message of length N is

$$T = \alpha + \beta N ,$$

where α represents a fixed startup overhead and β is the incremental transmission time per byte. Table 2 shows the startup and transmission time coefficients that were calculated from a least-squares fit of the echo data for single-hop messages for several hypercubes [3] and *dcube*. The large startup time for the *dcube* package reflects the high overhead of TCP/IP compared to the minimal protocols used on most hypercubes. Forwarding all messages through the *dcube* task further degrades communication performance.

Coefficients of Communication microseconds				
	Ametek	iPSC/1	Ncube	dcube(Sun)
Startup (α)	563.5	862.2	383.6	11,557.0
Byte transfer (β)	9.5	1.8	2.6	2.0

Table 2. *Least-squares estimate of communication coefficients.*

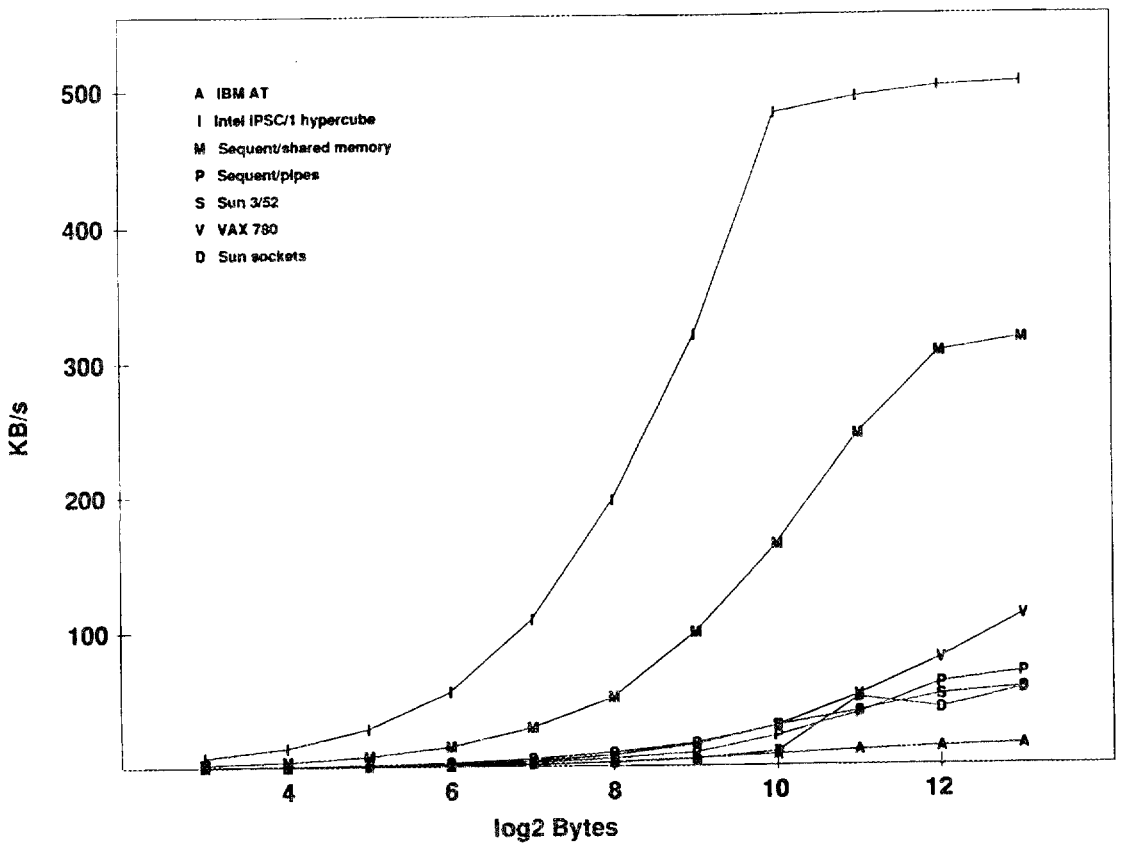

Figure 6. *Simulator and hypercube message-passing performance.*

Table 3 compares the Intel iPSC/1 and *dcube* implemented on a collection of Sun 3/50 workstations with respect to several metrics, including Cholesky factorization, numerical integration, computational speed, and communication speed. The Cholesky factorization is relatively fine grained and communication intensive, whereas the numerical integration is coarser grained and much less communication intensive. The large communication-to-computation ratio of the *dcube* package indicates that it is better suited to applications with heavy computation between message exchanges.

5. Future work

Although meeting the initial objectives, the computing environment provided by *dcube* is far from ideal. Message-passing performance could be improved by permitting direct process-to-process communication, rather than passing all messages through the *dcube* process; the trace file facility would be lost, but the user could still use *syslog()* for debugging. Communication could be further enhanced by taking advantage of the broadcast nature of Ethernet and token rings. Providing broadcast would require a protocol other the TCP, so further study of underlying protocols is warranted.

Metric	iPSC/1	Sun
Cholesky(4) (sec)	9	20
Cholesky(8) (sec)	6	16
Integration(8) (sec)	26	9
Integration(16) (sec)	13	5
Megaflops	0.04	0.16
Kilobytes/sec	504	46
8 byte comm. (μs)	1,120	12,200
8 byte multiply(μs)	43	17
comm./comp.	26	718

Table 3. *Hypercube and dcube performance on Sun workstations.*

The present implementation works only with a collection of homogeneous architectures; there is no provision for converting data to account for differing data representations (byte order, floating point representation, etc.) among differing architecures. A network file system simplifies program and data distribution, but heterogeneous architectures would still dictate multiple copies of the executables. The distributed program models provided by Sun (RPC/XDR) and Apollo (NCS) address these problems. The model is best implemented within a collection of machines under one administration, since issues of cost-recovery, ''using someone else's machine,'' and disk space have not been addressed.

The hypercube model does not accurately reflect the communication structure of most local area networks, so a different program model could be developed (see, e.g., [5]). Indeed, many of the applications and algorithms that we have ported from the hypercube environment perform poorly under *dcube* because these codes use message-passing structures that are optimized for a hypercube network. We also are investigating other promising distributed/parallel computing models such as Sprite [8] and MACH [10].

References

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

[2] T. H. Dunigan, *A Portable Hypercube Simulator*, Tech. Rept. ORNL/TM-10140, Oak Ridge National Laboratory, Oak Ridge, TN (1987).

[3] T. H. Dunigan, *Performance of Three Hypercubes*, Tech. Rept. ORNL/TM-10400, Oak Ridge National Laboratory, Oak Ridge, TN (1986).

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

[5] T. Gardner, I. Gerard, C. Moweers, E. Nemeth, and R. Schnabel, *DPUP: A distributed processing utilities package*, Computer Sci. Tech. Rept., Univeristy of Colorado, Boulder, CO (1986).

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

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

[8] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch, *The Sprite Network Operating System*, Computer, February 1988, pp. 23-36.

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

[10] A. Tevanian and R. F. Rashid, *MACH: A Basis for Future UNIX Devlopment*, Computer Science Dept. Tech. Rept., CMU, Pittsburgh, PA (1987).

Appendix A

Simulator Manual Pages

NAME

dcube - distributed hypercube simulator

SYNOPSIS

dcube commands:

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

The host and node programs should be linked with the library *libdcube.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 mypid();
int cubedim();
int clock();
void syslog(int pid, char *msg);
void flick();
```

DESCRIPTION

A simulator driver task, *dcube*, provides a set of commands for loading and controlling a set of hypercube application programs built with the simulator library *libdcube.a* and optionally distributed over a network of computers. The simulator can be used to execute programs developed for the Intel iPSC/1 hypercube, and a trace file is provided to assist in debugging or performance analysis. The simulator uses the UNIX fork, *rsh*, and TCP/IP facilities and has been tested on clusters of Sun workstations, IBM RT's, DEC VAX with UNIX 4.3, and a Sequent Balance 8000 with DYNIX. The various application processes that

constitute the hypercube simulation may be run on one machine or distributed over a network of machines. Since the simulation will spawn multiple processes when run on a single machine, you may wish to use *nice(1)* to reduce system load. When run on a single parallel-processor machine such as the Sequent or Encore, the simulation 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 *ppsim(l)*, but does permit porting of programs between the Intel cube and the simulator with no source code changes. With *dcube* one can develop applications taking advantage of the large virtual memory and I/O services available from a collection of remote machines. The programmer compiles and links his host and node programs with *libdcube.a* and then runs *dcube* 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. The *h* command (host) specifies an executable file that will be the application host program and, optionally, a remote machine on which the application is to be run. It is recommended that the host program be run on the same machine as *dcube* (omit the *-h* option) so that the host program may perform terminal input and output. The *l* command loads the simulator nodes with the program file. A specific node may be loaded with the *-n* option. The *-h* option can be used to specify a remote machine on which the node program(s) are to be run. The *-c* option specifies the dimension of the cube and must be in the range 0-4. The number of processes that can be controlled by *dcube* is limited by the maximum number of files a process may open. 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. Commands can be placed in a file and used with the *u* command. A sample session for running the programs on one machine might be

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

To run the node programs on a collection of machines on the network, one could construct the following command file, *sample.in*,

```
h host
l -c 2
l -n 0 -h mthsun node
l -n 1 -h crfsun node
l -n 2 -h puwsun node
l -n 3 -h yhesun node
```

then invoke *dcube* and use the following commands:

```
u sample.in
s
```

The functions provided in *libdcube.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 an integer descriptor 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 the Intel iPSC/1 hypercube.

recv and *recvw* await the arrival of a message of the given type for the node and process id associated with integer channel descriptor *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 executing 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 iPSC/1 hypercube. *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 channel descriptor *d*. 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 channel 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*, although *flick* does nothing in the distributed cube simulation.

Note that *dcube* uses one file descriptor (logical unit number 3) in managing the simulation through TCP sockets.

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.

FILES

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

libdcube.a	simulator subroutines
dcube	simulator driver
rmtdcube	remote simulator application initiator
dbld	sample C build script
dfbld	sample f77 build script

ASSUMPTIONS

It is assumed that *rmtdcube* is in the default execution path on the remote. It is sufficient for it to reside in the user's execution path. It is assumed that the executables to be run on the remote exist on the remote and that the user has accounts under the same name on all remotes. The various remotes should all be in the */etc/hosts.equiv* or the user's *.rhosts* file.

SEE ALSO

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

BUGS

The delay in message passing is due to delays in relaying the messages through TCP sockets and does not reflect a hypercube structure. Messages are passed through the simulator driving task *dcube*. The clock used is just the computer's time of day clock and typically has a resolution of only 20 milliseconds. Passing time-stamps between remote applications is not recommended since the clocks of the remote machines are not likely synchronized. Time stamps in the trace file are synchronized. Using a collection of heterogeneous machines as remotes may require additional application programming to account for difference in data representations (byte order, floating point format, etc.). When *dcube* terminates you may ignore the messages *nn: no such process*. The *handler* function and the Intel dynamic loader functions are not presently implemented.

AUTHOR

T. H. Dunigan

INTERNAL DISTRIBUTION

1-5. T. H. Dunigan
6-7. R. F. Harbison/ Mathematical Sciences Library
8. G. A. Geist
9. M. T. Heath
10. J. K. Ingersoll
11. J. M. Jansen
12. M. R. Leuze
13. F. C. Maienschein
14. E. G. Ng
15. B. W. Peyton
16. C. H. Romine
17-21. R. C. Ward
22. C. Weisbin
23. J. Wooten
24. P. H. Worley
25. A. Zucker
26. Central Research Library
27. K-25 Plant Library
28. ORNL Patent Office
29. Y-12 Technical Library/ Document Reference Section
30. Laboratory Records--RC
31-32. Laboratory Records Department
33. J. J. Dorning (Consultant)
34. R. W. Haralick (Consultant)

EXTERNAL DISTRIBUTION

35. Dr. Donald M. Austin, Office of Scientific Computing, Office of Energy Research, ER-7, Germantown Building, U.S. Department of Energy, Washington, DC 20545

36. Dr. Jesse L. Barlow, Department of Computer Science, Pennsylvania State University, University Park, PA 16802

37. Dr. Bill L. Buzbee, C-3, Applications Support & Research, Los Alamos National Laboratory, P.O. Box 1663, Los Alamos, NM 87545

38. Dr. Donald A. Calahan, Department of Electrical and Computer Engineering, University of Michigan, Ann Arbor, MI 48109

39. Dr. John Cavallini, ER-7, GTN, Office of Scientific Computing, Department of Energy, Washington, DC 20545

40. Dr. Jagdish Chandra, Army Research Office, P.O. Box 12211, Research Triangle Park, NC 27709

41. Dr. Melvyn Ciment, National Science Foundation, 1800 G Street N.W., Washington, DC 20550

42. Dr. George Cybenko, Department of Mathematics, Tufts University, Medford, MA 02155

43. Dr. George J. Davis, Department of Mathematics, Georgia State University, Atlanta, GA 30303

44. Dr. Jack J. Dongarra, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439

45. Dr. Geoffrey C. Fox, Physics Department, California Institute of Technology, Pasadena, CA 91125

46. Dr. Paul O. Frederickson, Computing Division, Los Alamos National Laboratory, Los Alamos, NM 97545

47. Dr. Dennis B. Gannon, Computer Science Department, Purdue University, West Lafayette, IN 47907

48. Dr. David M. Gay, Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974

49. Dr. W. Morven Gentleman, Division of Electrical Engineering, National Research Council, Building M-50, Room 344, Montreal Road, Ottawa, Ontario, Canada K1A OR8

50. Dr. Jung Hong, Los Alamos National Laboratory, P.O. Box 1663, MS K488, Los Alamos, NM 87545

51. Dr. Don E. Heller, Physics and Computer Science Department, Shell Development Co., P.O. Box 481, Houston, TX 77001

52. Prof. Roger W. Hockney, Department of Computer Science, University of Reading, Whiteknights, Bersk., England RG6 2AX

53. Dr. Robert E. Huddleston, Computation Department, Lawrence Livermore National Laboratory P.O. Box 808, Livermore, CA 94550

54. Dr. Ilse Ipsen, Department of Computer Science, Yale University, P.O. Box 2158 Yale Station, New Haven, CT 06520

55. Dr. Harry Jordan, Department of Electrical and Computer Engineering, University of Colorado, Boulder, CO 80309

56. Dr. Robert J. Kee, Applied Mathematics Division 8331, Sandia National Laboratories, Livermore, CA 94550

57. Ms. Virginia Klema, Statistics Center, E40-131, MIT, Cambridge, MA 02139

58. Dr. Richard Lau, Office of Naval Research, 1030 E. Green Street, Pasadena, CA 91101

59. Dr. Alan J. Laub, Department of Electrical and Computer Engineering, University of California, Santa Barber, CA 93106

60. Dr. Robert L. Launer, Army Research Office, P.O. Box 12211, Research Triangle Park, NC 27709

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

62. Dr. Olaf Lubek, C-3, Computer Research and Applications, Los Alamos National Laboratory P.o. Box 1663 Los Alamos, NM 87545

63. Dr. Franklin Luk, Electrical Engineering Department, Cornell University, Ithaca, NY 14853

64. Dr. Paul C. Messina, Applied Mathematics Division, Argonne National Laboratory, Argonne, IL 60439

65. Mr. Bob Micci, Sequent Computer Systems, Inc., 400 Amherst St., Nashua, NH 03063

66. Dr. George Michael, Computation Department, Lawrence Livermore National Laboratory, P.O. Box 808, Livermore, CA 94550

67. Dr. Dianne P. O'Leary, Computer Science Department, University of Maryland, College Park, MD 20742

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

69. Dr. Neil Ostlund, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N3L 3G1

70. Dr. Edward W. Page, Dept. of Computer Science, 405 College of Nursing Bldg., Clemson, SC 29634-1906

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

72. Prof. Merrell Patrick, Department of Computer Science, Duke University, Durham, NC 27706

73. Dr. Robert J. Plemmons, Department of Mathematics and Computer Science, North Carolina State University, Raleigh, NC 27650

74. Dr. John R. Rice, Computer Science Department, Purdue University, West Lafayette, IN 47907

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

76. Dr. Donald J. Rose, Department of Computer Science, Duke University, Durham, NC 27706

77. Dr. Ahmed H. Sameh, Computer Science Department, University of Illinois, Urbana, IL 61801

78. Dr. Joel Saltz, ICASE, MS 132-C, NASA, Langley, Hampton, VA 23665

79. Dr. Robert Schreiber, Dept. of Mathematical Sciences, Rensselaer Polytechnic Institute, Troy, NY 12180

80. Dr. Martin H. Schultz, Department of Computer Science, Yale University, P.O. Box 2158 Yale Station, New Haven, CT 06520

81. Dr. David S. Scott, Intel Scientific Computers, 15201 N.W. Greenbrier Parkway, Beaverton, OR 97006

82. Dr. Danny C. Sorensen, Mathematics and Computer Science Div., Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439

83. Prof. G. W. Stewart, Computer Science Department, University of Maryland, College Park, MD 20742

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

85. Prof. Charles Van Loan, Department of Computer Science, Cornell University, Ithaca, NY 14853

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

87. Dr. Andrew B. White, Computing Division, Los Alamos National Laboratory, Los Alamos, NM 87545

88. Dr. Arthur Wouk, Army Research Office, P.O. Box 12211 Research Triangle Park, NC 27709

89. Dr. Margaret Wright, Systems Optimization Laboratory, Operations Research Department, Stanford University, Stanford, CA 94305

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

91-100. Office of Scientific & Technical Information, P. O. Box 62, Oak Ridge, TN 37831