

MARTIN MARIETTA ENERGY SYSTEMS LIBRARIES



3 4456 0316552 6

ORNL/TM-11616

oml

**OAK RIDGE
NATIONAL
LABORATORY**

MARTIN MARIETTA

A Users' Guide to PICL

**A Portable Instrumented
Communication Library**

G. A. Geist
M. T. Heath
B. W. Peyton
P. H. Worley

OAK RIDGE NATIONAL LABORATORY
CENTRAL RESEARCH LIBRARY
CIRCULATION SECTION
ALBION ROOM 111
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

This report has been reproduced directly from the best available copy.

Available to DCE and DCE contractors from the Office of Scientific and Technical Information, P.O. Box 62, Oak Ridge, TN 37831; prices available from (615) 576-8401, FTS 826-8401.

Available to the public from the National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Rd., Springfield, VA 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-11616

Engineering Physics and Mathematics Division

Mathematical Sciences Section

**A USERS' GUIDE TO PICL
A PORTABLE INSTRUMENTED COMMUNICATION LIBRARY**

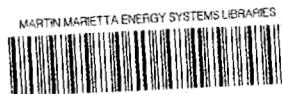
G. A. Geist
M. T. Heath
B. W. Peyton
P. H. Worley

Oak Ridge National Laboratory
Mathematical Sciences Section
P.O. Box 2009, Bldg. 9207-A
Oak Ridge, TN 37831-8083

Date Published: October 1990

Research was supported by the Applied Mathematical Sciences Research Program 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-AC-05-84OR21400



3 4456 0316552 6

Contents

1	Introduction	1
2	Low-Level Routines	2
3	High-Level Routines	6
4	Tracing Routines	10
5	Appendix	15
	5.1 Fortran Examples	15
	5.2 Obtaining PICL	21
	5.3 Obtaining ParaGraph	22
6	References	22

**A USERS' GUIDE TO PICL
A PORTABLE INSTRUMENTED COMMUNICATION LIBRARY**

G. A. Geist
M. T. Heath
B. W. Peyton
P. H. Worley

Abstract

This report is the PICL user's guide. It contains an overview of PICL and how it is used. Examples in C and Fortran are included.

PICL is a subroutine library that can be used to develop parallel programs that are portable across several distributed-memory multiprocessors. PICL provides a portable syntax for key communication primitives and related system calls. It also provides portable routines to perform certain widely-used, high-level communication operations, such as global broadcast and global summation. Finally, PICL provides execution tracing that can be used to monitor performance or to aid in debugging.

1. Introduction

PICL is a portable instrumented communication library designed to provide portability, ease of programming, and execution tracing in parallel programs.

PICL provides portability between many machines and multiprocessor environments. It is fully implemented on the Intel iPSC/2, the Intel iPSC/860, and the Ncube/3200 families of hypercube multiprocessors and on the Cogent multiprocessor workstation. A subset of the library is provided for each of the following distributed-memory multiprocessors and multiprocessor programming environments: the Intel iPSC/1, the Symult S2010, the Cosmic Environment, Linda, Unix System V, and the X Window System. Full implementations of the library will be available on most of these target machines and environments in the near future. The list is expected to grow as new machines and programming environments appropriate for the library appear, such as the Ncube/6400.

In addition to supplying low-level communication primitives, such as *send* and *receive*, PICL simplifies parallel programming by providing a set of high-level communication routines such as global broadcast, global maximum, and barrier synchronization. These routines can help the novice user avoid common synchronization and programming errors and save programming time even for the veteran user. These high-level routines also facilitate experimentation and performance optimization by supporting a variety of interconnection topologies.

Execution tracing has been built into the PICL routines, and routines are provided to control the type and amount of tracing. A separate package called ParaGraph [2] is available to display the tracing output graphically. The tracing facility is useful for performance modeling, performance tuning, and debugging.

This document contains examples and information needed for straightforward use of most of PICL's basic features. Full documentation of all PICL options and the various ways the library can be used is contained in a separate report [1].

The library is made up of three distinct sets of routines: a set of low-level communication and system primitives described in section 2, a set of high-level global communication routines whose use is described in section 3, and a set of routines for invoking and controlling the execution tracing facility, which is described in section 4. Each section contains examples in C showing typical uses of the respective routines. In

addition, the Appendix contains FORTRAN versions of the examples and instructions for obtaining PICL and ParaGraph.

2. Low-Level Routines

The 12 low-level communication and system interface routines, described in Table 1, provide a portable syntax for message-passing programs.

The PICL programming model assumes that the multiprocessor can send messages between arbitrarily chosen pairs of processors. The time required to send a message between two processors is a function of the interprocessor communication network, and a user will need to be aware of such machine dependencies in order to write efficient programs. Our model distinguishes one processor, the host, from the rest. The user has access to the remaining processors, called node processors (or simply nodes), through the host. Typically, an application code consists of one program that runs on the host and another program that runs on each of the nodes. The host program calls PICL routines to allocate node processors, load the node program (or programs) onto the nodes, send input data required by the node programs, and receive results from the nodes.

Figures 1 and 2 give a template of typical host and node programs that use only low-level PICL routines. The host must call `open0` to allocate nodes and enable interprocessor communication. It then must call `load0` to load the node program(s) onto the processors. The node program must also call `open0` to enable interprocessor communication. Subsequently, `send0` and `recv0` are used to pass messages between processors.

Both host and node programs must call `close0` to disable interprocessor communication. On the node, `close0` must be the last executable statement. On the host, `close0(1)` waits until all the nodes have executed `close0` and then releases the allocated nodes. The routines `clock0` and `check0` have the distinction that they can be called outside the bracket of `open0` and `close0`. All other low-level routines generate an error message and cause the program to terminate if called before `open0` or after `close0`.

In our programming model, a program returns from the `send0` command as soon as the user's message buffer can be reused safely, even if the message has not yet

void check0(int checking) - disables parameter checking if <code>checking = 0</code> , and enables parameter checking in PICL routines if <code>checking = 1</code> . By default, parameter checking is enabled.
double clock0() - returns the local system clock time in seconds.
void close0(int release) - disables interprocessor communication. On the host, allocated processors are also released if <code>release = 1</code> .
void load0(char *file, int node) - loads a program on node number <code>node</code> . If <code>node = -1</code> , then the program is loaded on every node. (<i>Host only</i>)
void message0(char *message) - prints a short message (≤ 80 characters) on the standard output device of the host.
void open0(int *numproc, int *me, int *host) - On the host, <code>open0</code> allocates <code>numproc</code> node processors, enables interprocessor communication, and returns the host's ID number. On a node, <code>open0</code> enables interprocessor communication and returns the number of allocated processors, the node's ID number, and the host's ID number.
int probe0(int msgtype) - returns the value 1 if a message of the specified type has arrived, and returns 0 otherwise. If <code>msgtype = -1</code> , then <code>probe0</code> checks for messages of any type. (<i>Nonblocking</i>)
void recv0(char *buf, int bytes, int msgtype) - receives a message of the specified type into a buffer of size <code>bytes</code> (in bytes). If <code>msgtype = -1</code> , then any type is accepted. (<i>Blocking</i>)
void recvinfo0(int *bytes, int *msgtype, int *source) - returns information about the most recent <code>recv0</code> or successful <code>probe0</code> call.
void send0(char *buf, int bytes, int msgtype, int dest) - sends a message of length <code>bytes</code> (in bytes) to processor number <code>dest</code> . The <code>msgtype</code> must be ≥ 0 .
void sync0() - executes barrier synchronization of all allocated processors. (<i>Node only</i>)
void who0(int *numproc, int *me, int *host) - returns the number of allocated node processors, the ID number of the processor calling <code>who0</code> , and the ID number of the host.

Table 1: PICL Low-Level Primitives

```
main()
{
    int nproc, me, host, bytes, msgtype, node, ...
    double time[2], result, data[100], clock0(), ...

    time[0] = clock0() ;
    /* Allocate 32 processors and enable PICL communication */
    nproc = 32 ;
    open0( &nproc, &me, &host ) ;

    /* Load node program onto all nodes */
    load0( "nodeprogram", -1 ) ;

    /* ----- Begin user program ----- */
    .
    .
    bytes = sizeof(data) ;
    msgtype = 1 ;
    node = 0 ;
    send0( data, bytes, msgtype, node ) ;
    .
    .
    /* wait for results from nodes */
    bytes = sizeof(result) ;
    msgtype = 2 ;
    recv0( &result, bytes, msgtype ) ;
    .
    .
    time[1] = clock0() ;
    printf( "host took %f seconds to finish", time[1]-time[0] ) ;
    .
    /* ----- End user program ----- */

    /* release allocated processors and disable communication */
    close0(1) ;
}
```

Figure 1: Host program template using only low-level routines.

```
main()
{
    int nproc, me, host, bytes, msgtype, node, ...
    double time, result, data[100], clock0(), ...

    /* Enable PICL communication */
    open0( &nproc, &me, &host ) ;

    /* ----- Begin user program ----- */
    .
    .
    /* Receive data from host and distribute */
    bytes = sizeof(data) ;
    msgtype = 1 ;
    if( me == 0 )
    {
        recv0( data, bytes, msgtype ) ;
        for( i=1 ; i<nproc ; i++ ) send0( data, bytes, msgtype, i ) ;
    }
    else
        recv0( data, bytes, msgtype ) ;
    .
    .
    time = clock0() ;
    user_routine() ;
    result = clock0()-time ;

    /* Send timing result to host */
    if( me == 0 )
    {
        bytes = sizeof(result) ;
        msgtype = 2 ;
        send0( &result, bytes, msgtype, host ) ;
    }
    .
    .
    /* ----- End user program ----- */

    /* Disable PICL communication */
    close0() ;
}
```

Figure 2: Node program template using only low-level routines.

arrived at the destination processor. On the receiving end, the processor is idle (or blocked) from the time it issues the `recv0` command until a message satisfying the request arrives and is copied into the specified user buffer. Note that a program will not terminate if a `recv0` command is never satisfied by an arriving message of the correct type. Moreover, only the type field distinguishes different messages in PICL. A common mistake made by new users is not using enough distinct types in their `send0` and `recv0` calls to uniquely identify different messages in their program. This often leads to nondeterministic behavior of the user's algorithm.

3. High-Level Routines

The high-level routines, which are built on top of the low-level routines, are global communication functions that have proven useful in the development of parallel algorithms and application programs. Users who require an unsupplied variant or generalization of one or more of the high-level routines in the library should be able to save a significant amount of work and obtain portability by modeling the new routine on the corresponding library routines.

The high-level routines, summarized in Table 3, are designed to run on various network topologies so that the user can take advantage of the physical interconnection network and algorithm characteristics. The routine `setarc0` must be called by the host and nodes before any of the other high-level routines, as illustrated in Figures 3 and 4. The host's `setarc0` sets the architectural parameters to be used by the high-level routines, while the node's call to `setarc0` retrieves these parameters, which are:

- `nprocs` - the number of processors in use. It must be between 0 and the number of nodes allocated by `open0` on the host.
- `top` - topology flag, where 1=hypercube, 2=full connectivity, 3=unidirectional ring, 4=bidirectional ring.
- `ord` - numbering of the nodes in a ring, where 0=natural ordering, i.e., 0, 1, 2, 3, ... and 1=Gray code ordering, i.e., 0, 1, 3, 2, ...
- `dir` - direction of broadcast in unidirectional ring, where 1=forward (i.e., from lower to higher numbered nodes) and -1 =backward.

The use of hypercube topology or Gray order requires that `nprocs` be a power of two. The routine `setarc0` can be called several times during a single run to vary the topology

<code>void barrier0()</code> - executes barrier synchronization of all nodes specified by <code>setarc0</code> .
<code>void bcast0(char *buf, int bytes, int msgtype, int root)</code> - broadcasts a message.
<code>void gand0(char *buf, int items, int datatype, int msgtype, int root)</code> - computes the componentwise "AND" of a distributed set of vectors. Datatypes for logical operations are: 0=char, 1=short, 2=int, 3=long.
<code>void gcomb0(char *buf, int items, int datatype, int msgtype, int root, void (*comb)())</code> - computes a user-defined componentwise combination of a distributed set of vectors. Datatypes for arithmetic operations are: 0=char, 1=short, 2=int, 3=long, 4=float, 5=double.
<code>void getarc0(int *nprocs, int *top, int *ord, int *dir)</code> - returns the number of processors and architecture parameters specified by the most recent call to <code>setarc0</code> .
<code>int ginv0(int i)</code> - returns the inverse binary reflected Gray code of <code>i</code> .
<code>void gmax0(char *buf, int items, int datatype, int msgtype, int root)</code> - computes the componentwise maximum of a distributed set of vectors.
<code>void gmin0(char *buf, int items, int datatype, int msgtype, int root)</code> - computes the componentwise minimum of a distributed set of vectors.
<code>void gor0(char *buf, int items, int datatype, int msgtype, int root)</code> - computes the componentwise "OR" of a distributed set of vectors.
<code>void gprod0(char *buf, int items, int datatype, int msgtype, int root)</code> - computes the componentwise product of a distributed set of vectors.
<code>int gray0(int i)</code> - returns the binary reflected Gray code of <code>i</code> .
<code>void gsum0(char *buf, int items, int datatype, int msgtype, int root)</code> - computes the componentwise sum of a distributed set of vectors.
<code>void gxor0(char *buf, int items, int datatype, int msgtype, int root)</code> - computes the componentwise exclusive "OR" of a distributed set of vectors.
<code>void setarc0(int *nprocs, int *top, int *ord, int *dir)</code> - sets the number of processors and the interconnection topology to be used by the high-level communication routines.

Table 2: PICL High-Level Communication Routines

```
main()
{
  int nproc, me, host, bytes, datatype, msgtype, node, ...
  float results[100], ...

  nproc = 32 ;
  open0( &nproc, &me, &host ) ;
  load0( "nodeprogram", -1 ) ;

  /* set architectural parameters used by high level routines. */
  top = 1 ; /* set topology to hypercube */
  ord = 1 ; /* set node order to gray code */
  dir = 1 ; /* set ring direction to forward */
  setarc0( &nproc, &top, &ord, &dir ) ;

  /* ----- Begin user program ----- */
  .
  .
  /* Broadcast problem size */
  n = 100 ;
  bytes = sizeof(n) ;
  msgtype = 0 ;
  bcast0( &n, bytes, msgtype, host ) ;
  .
  .
  /* Collect global sum of node's results */
  datatype = 4 ; /* set data type to float */
  msgtype = 10 ;
  gsum0( results, n, datatype, msgtype, host ) ;
  .
  .
  /* ----- End user program ----- */

  close0(1) ;
}
```

Figure 3: Host program template using high- and low-level routines.

```
main()
{
    int nproc, me, host, bytes, datatype, msgtype, top, ord, dir, n, ...
    int vec[100], ...
    float results[100], ...

    open0( &nproc, &me, &host ) ;

    /* get architectural parameters used by high level routines. */
    setarc0( &nproc, &top, &ord, &dir ) ;

    /* ----- Begin user program ----- */
    .
    .
    /* Receive and participate in broadcast of problem size */
    bytes = sizeof(n) ;
    msgtype = 0 ;
    bcast0( &n, bytes, msgtype, host ) ;
    .
    .
    /* Collect global maximum of vec and broadcast result to all nodes */
    datatype = 2 ;          /* set datatype to int */
    msgtype = datatype ;
    root = 0 ;
    gmax0( vec, n, datatype, msgtype, root ) ;
    bytes = n*sizeof(float) ;
    bcast0( vec, bytes, msgtype, root ) ;
    .
    .
    /* Participate in global sum sending results to host */
    datatype = 4 ;          /* set datatype to float */
    msgtype = 10 ;
    gsum0( results, n, datatype, msgtype, host ) ;
    .
    .
    /* ----- End user program ----- */

    close0() ;
}
```

Figure 4: Node program template using high- and low-level routines.

and number of processors. A node will not return from `setarc0` until `nprocs` is either 0 or greater than its node ID. (The special case `nprocs= 0` is intended to be used as an “end of run” flag.)

To operate correctly, the high-level routines (with the exception of `gray0` and `ginv0`) must be called by all the nodes in use. For example, to broadcast a vector, `vec`, of size `bytes` from node 5 to the other nodes in use, all the nodes specified by `setarc0` would call

```
bcast0( vec, bytes, type, 5 ) ;
```

On return, every node's `vec` would match `vec` on node 5.

All the nodes must know either implicitly or from a previous message the root of a particular high-level call. For example, in Figure 4 all the nodes call `gmax0` with `root= 0`. After this, only node 0 knows the maxima, so all the nodes then call `bcast0` with `root= 0`, after which they all know the maxima. Figures 3 and 4 also illustrate the use of the host as the root of a high-level call.

4. Tracing Routines

When the user requests execution tracing, code is activated within PICL routines that produces time-stamped records detailing the course of the computation on each processor. One of the key quantities captured is the time each processor spends blocked while waiting for messages from other processors. With this and similar data, the user can evaluate the performance of his code and locate possible performance bottlenecks. Execution tracing is controlled by the routines described in Table 3.

Assuming the user wishes to trace only the execution on the nodes (host execution tracing is also possible but seldom used), three tracing routines are required: `traceenable` on the host, and `tracelevel` and `tracenode` on the node(s). Examples of their use can be seen in Figures 5 and 6. `Traceenable` is typically the first executable statement in the host program. This routine specifies the name of the trace file, enables tracing, and sets the output format used for the trace records. If `format = 1`, then keywords are inserted into each trace record to help the user read the trace file. If `format = 0`, then the trace records are written out as a compact set of integers that can then be input into the ParaGraph package for display after proper sorting.

void traceenable(char *tracefile, int format) <ul style="list-style-type: none">- opens a trace file and sets the trace record format. Set <code>format = 0</code> for use with ParaGraph. Set <code>format = 1</code> to label trace records with keywords. (<i>Host only</i>)
void traceexit() <ul style="list-style-type: none">- stops tracing.
void traceflush() <ul style="list-style-type: none">- sends local trace information to the trace file.
void tracehost(int tracesize, int flush) <ul style="list-style-type: none">- starts tracing on the host and specifies how large a buffer to reserve for trace information. If <code>flush = 1</code>, then <code>traceflush</code> is called automatically if the trace buffer fills up. If <code>flush = 0</code>, then tracing stops if the trace buffer fills up. (<i>Host only</i>)
void traceinfo(int remaining, int event, int compstats, int commstats) <ul style="list-style-type: none">- returns the current tracing parameters, as set by <code>tracelevel</code>, and an estimate of the number of trace records that will fit in the remaining space in the trace buffer.
void tracelevel(int event, int compstats, int commstats) <ul style="list-style-type: none">- sets the parameters that control the amount of trace data generated for three types of trace records. A zero value denotes the least amount of information generated, while ≥ 3 denotes the most.
void tracemark(int marktype) <ul style="list-style-type: none">- generates a trace record marking a user-specified event.
void tracemsg(char *message) <ul style="list-style-type: none">- sends a short message (≤ 80 characters) to the host that will automatically be written into the trace file.
void tracenode(int tracesize, int flush, int sync) <ul style="list-style-type: none">- starts tracing on a node and specifies how large a buffer to reserve for trace information. If <code>flush = 1</code>, then <code>traceflush</code> is called automatically if the trace buffer fills up. If <code>flush = 0</code>, then tracing stops if the trace buffer fills up. If <code>sync = 1</code>, then node system clocks are synchronized before tracing begins. (<i>Node only</i>)

Table 3: PICL Execution Tracing Routines

The following Unix command performs this sort:

```
sort +1n -2 +2n -3 +0n -1 tracefile > ParaGraph.input
```

The routine `tracenode` is called by the node program to create a local trace buffer and start tracing. The synchronization option (`sync = 1`) should be used for trace files that are to be analyzed with ParaGraph, and all nodes must call `tracenode` with `sync` set to this value. The effect of setting `sync` is to synchronize node clocks so that time stamps will be consistent. Thus, the user should be careful where `tracenode` is called because it entails a barrier synchronization. If trace information is needed on only one or a few nodes (for example, to do debugging), then these nodes must call `tracenode` with `sync = 0`, and the other nodes should not call `tracenode`.

There are four distinct types of trace records generated: event, computation statistics, communication statistics, and trace message. The routine `tracelevel` sets the amount of trace data generated for the first three trace record types. The value of `event` determines which PICL routines will generate “event” trace records. If `event = 0`, then only calls to `open0`, `close0`, `tracelevel`, `tracenode`, `traceflush`, `traceexit`, and `tracehost` are recorded. If `event = 1`, then records are also generated for `tracemark`. If `event = 2`, then records are also generated for the high-level routines as well as `send0` and `recv0` events outside the high level routines. Finally, if `event ≥ 3`, then records are also generated for `send0` and `recv0` events embedded inside the high-level routines. The values for `compstats` and `commstats` similarly control which events generate statistical records.

When the tracing logic is not activated, there is very little overhead incurred by using the PICL routines *on the nodes*. Due to the possibility of tracing, `recv0 on the host` is two to three times slower than the native command. Since the host tends to be significantly slower than the nodes even when not using PICL, most application codes should minimize the use of the host.

If the tracing logic is enabled and the trace information is sent to the trace file at arbitrary points in the node (or host) programs, then the additional cost due to tracing can be very high, and might cause an unacceptable perturbation in the behavior of the program under study. This can occur when `flush = 1` in the call to `tracenode` (or `tracehost`). If `flush = 0` and `traceflush` is not used, then trace information is sent back only at the end of the node program, and the resulting cost is quite reasonable. To

avoid the need for large trace buffers, it is recommended that `tracelevel` be used as shown in Figure 6 to trace only those portions of the code in which the user is interested. This has the added benefit of reducing the size of the trace files. ParaGraph displays can be improved by positioning the `tracenode` call near the point of interest. This avoids long blank displays between the call to `tracenode` and the interesting part of the code.

```
main()
{
    int nproc, me, host, bytes, type, top, ord, dir, ...
    double x, ...

    /* Open tracefile and use compact ParaGraph format */
    traceenable( "tracefile", 0 ) ;

    nproc = 8 ;
    open0( &nproc, &me, &host ) ;
    load0( "nodeprogram", -1 ) ;

    top = 3 ; /* set topology to ring          */
    ord = 0 ; /* set node order to natural     */
    dir = -1 ; /* set ring direction to backward */
    setarc0( &nproc, &top, &ord, &dir ) ;

    /* ----- Begin user program ----- */
    .
    .
    .
    recv0( x, sizeof(x), 3 ) ;
    /* ----- End user program ----- */

    close0(1) ;
}
```

Figure 5: Host program template for tracing node execution.

```
main()
{
  int nproc, me, host, bytes, datatype, msgtype, top, ord, dir, n, ...
  double x, ...

  /* Start tracing using 100K local buffer */
  tracenode( 100000, 0, 1 ) ;
  tracelevel( 0, 0, 0 ) ;

  .
  .
  open0( &nproc, &me, &host ) ;
  setarc0( &nproc, &top, &ord, &dir ) ;

  /* ----- Begin user program ----- */
  .
  .
  /* Set trace levels to typical values used for ParaGraph) */
  tracelevel( 4, 4, 0 ) ;
  .
  .
  /* Calculate global product with result on node nproc-1 */
  datatype = 5 ;          /* set datatype to double */
  msgtype = 1 ;
  root = nproc-1 ;
  gprod0( &x, 1, datatype, msgtype, root ) ;

  /* Turn off tracing for uninteresting sections */
  tracelevel( 0, 0, 0 ) ;
  .
  .
  if(me == nproc-1) send0( x, sizeof(x), 3, host ) ;
  /* ----- End user program ----- */

  /* Stop tracing */
  traceexit() ;

  /* close0 flushes local buffer to tracefile on host */
  close0() ;
}
```

Figure 6: Node program template with tracing.

5. Appendix

5.1. Fortran Examples

```
integer nproc, me, host, bytes, msgtype, node, ...
double precision time(2), result, data(100), clock0, ...

time(0) = clock0()
c Allocate processors and enable PICL communication
nproc = 32
call open0( nproc, me, host )

c Load node program onto all nodes
call load0( "nodeprogram", -1 )
c ----- Begin user program -----
.
.
bytes = n*8
msgtype = 1
node = 0
call send0( data, bytes, msgtype, node )
.
.
c wait until nodes are finished
bytes = 8
msgtype = 2
call recv0( result, bytes, msgtype )

time(1) = clock0()
print *, "host took", time(1)-time(0), " seconds to finish"
.
c ----- End user program -----
c release allocated processors and disable communication
call close0(1)
stop
end
```

Figure 7: Fortran host program template using only low-level routines.

```
integer nproc, me, host, bytes, msgtype, node, ...
double precision time, result, data(100), clock0(), ...

c  Enable PICL communication
    call open0( nproc, me, host )

c  ----- Begin user program -----
    .
    .
c  Receive data from host and distribute
    bytes = n*8
    msgtype = 1
    if( me .eq. 0 ) then
        call recv0( data, bytes, msgtype )
        do 10 i=1, nproc-1
            call send0( data, bytes, msgtype, i )
10    continue
        else
            call recv0( data, bytes, msgtype )
        endif
    .
    .
    time = clock0()
    call user_routine()
    result = clock0()-time

c  Send results to host
    if( me .eq. 0 ) then
        bytes = 8
        msgtype = 2
        call send0( result, bytes, msgtype, host )
    endif
    .
    .
c  ----- End user program -----

c  Disable PICL communication
    call close0()
    stop
end
```

Figure 8: Fortran node program template using only low-level routines.

```
integer nproc, me, host, bytes, datatype, msgtype, node, ...
real results(100), ...

nproc = 32
call open0( nproc, me, host )
call load0( "nodeprogram", -1 )

c  set architectural parameters used by high level routines.
   top = 1
   ord = 1
   dir = 1
   call setarc0( nproc, top, ord, dir )

c  ----- Begin user program -----
   .
   .
c  Broadcast problem size
   n = 100
   bytes = 4
   msgtype = 0
   call bcast0( n, bytes, msgtype, host )
   .
   .
c  Collect global sum of node's results
   datatype = 4
   msgtype = 10
   call gsum0( results, n, datatype, msgtype, host )
   .
   .
c  ----- End user program -----

   call close0(1)
   stop
   end
```

Figure 9: Fortran host program template using high- and low-level routines.

```
integer nproc, me, host, bytes, datatype, msgtype
integer top, ord, dir, n, vec(100), ...
real results(100), ...

call open0( nproc, me, host )

c  get architectural parameters used by high level routines.
   call setarc0( nproc, top, ord, dir )

c  ----- Begin user program -----
      .
      .
c  Receive and participate in broadcast of problem size
   bytes  = sizeof(int)
   msgtype = 0
   call bcst0( n, bytes, msgtype, host )
      .
      .
c  Collect global maximum of vec and broadcast result to all nodes
   datatype = 2
   msgtype  = n+5
   root     = 0
   call gmax0( vec, n, datatype, msgtype, root )
   bytes = n*4
   call bcst0( vec, bytes, msgtype, root )
      .
c  Participate in global sum sending results to host
   datatype = 4
   msgtype  = 10
   call gsum0( results, n, datatype, msgtype, host )
      .
      .
c  ----- End user program -----

   call close0()
   stop
   end
```

Figure 10: Fortran node program template using high- and low-level routines.

```
integer nproc, me, host, top, ord, dir, ...
double precesion x, ...

c  Open tracefile and use compact ParaGraph format
    call traceenable( "tracefile", 0 )

nproc = 8
call open0( nproc, me, host )
call load0( "nodeprogram", -1 )

top = 3
ord = 0
dir = -1
call setarc0( nproc, top, ord, dir )

c  ----- Begin user program -----
    .
    .
    .
    call recv0( x, 8, 3 )
c  ----- End user program -----

call close0(1)
stop
end
```

Figure 11: Fortran host program template for tracing node execution.

```
integer nproc, me, host, bytes, datatype, msgtype
integer top, ord, dir, n, ...
double precision x, ...

c Start tracing using 100K local buffer
  call tracenode( 100000, 0, 1 )
  call tracelevel( 0, 0, 0 )

  .
  call open0( nproc, me, host )
  call setarc0( nproc, top, ord, dir )

c ----- Begin user program -----
  .
  .
c Set trace levels to typical values used for ParaGraph)
  call tracelevel( 4, 4, 0 )
  .
  .
c Calculate global product with result on node nproc-1
  datatype = 5
  msgtype = 1
  root = nproc-1
  call gprod0( x, 1, datatype, msgtype, root )

c Turn off tracing for uninteresting sections
  call tracelevel( 0, 0, 0 )
  .
  .
  if(me .eq. nproc-1) call send0( x, 8, 3, host )
c ----- End user program -----

c Stop tracing
  call traceexit()

c close0 flushes local buffer to tracefile on host
  call close0()
  stop
  end
```

Figure 12: Fortran node program template with tracing.

5.2. Obtaining PICL

The source code for PICL is available from **netlib**. The PICL source is written in C, but Fortran-to-C interface routines are also supplied on those machines where it is feasible. Currently, **netlib** contains the following files:

picl.shar	low-level primitives and execution tracing routines
port.shar	high-level communication routines
cogent.shar	machine-dependent routines for the Cogent, including FORTRAN-to-C interface routines
ipsc2.shar	machine-dependent routines for the iPSC/2, including FORTRAN-to-C interface routines
ipsc860.shar	machine-dependent routines for the iPSC/860, including FORTRAN-to-C interface routines
ncube3200.shar	machine-dependent routines for the Ncube/3200, but without FORTRAN-to-C interface routines
userguide.shar	latex source of the PICL user guide (this document)
creference.shar	latex source of the reference manual for the C version of PICL.

More machine-dependent code will be added to this list in the near future.

To create PICL, you need the following *shar* files from the **picl** subdirectory on **netlib**: **picl.shar**, **port.shar**, and the appropriate machine-dependent code. Unpack all three in the same (empty) directory. A **README** file describing how to create the library is bundled with the machine-dependent *shar* file. For example, to get the source code for creating an iPSC/2 version of PICL, send the following message to **netlib@ornl.gov**:

```
send picl.shar from picl
send port.shar from picl
send ipsc2.shar from picl
```

The source code will arrive as one or more messages per *shar* file. Each message will contain a header describing what to remove and how to concatenate messages in order to recover a legal *shar* file. Once these instructions are done, the following in an empty directory:

```
sh picl.shar
```

```
sh port.shar
sh ipsc2.shar
```

You will now have a file `README`, a file `makefile`, and three subdirectories: `picl`, `port`, and `ipsc2`. The `README` file discusses how to compile the PICL routines and how to make the libraries `hostlib.a` and `nodelib.a`.

5.3. Obtaining ParaGraph

ParaGraph is also available from `netlib`. For information about this package send the following message to `netlib@ornl.gov`.

```
send index from paragraph
```

A short description of ParaGraph and instructions on how to build the package will be returned.

The source files are available in the *shar* file `paragraph.shar`. To receive this file send the message:

```
send paragraph.shar from paragraph
```

6. References

- [1] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. PICL: A portable instrumented communication library, C reference manual. Technical report, Oak Ridge National Laboratory, July 1990. ORNL/TM-11130.
- [2] M. T. Heath. Visual animation of parallel algorithms for matrix computations. In D. Walker, editor, *Proceedings of the Fifth Distributed Memory Computing Conference*. IEEE, 1990.

ORNL/TM-11616

INTERNAL DISTRIBUTION

- | | | | |
|--------|-------------------|--------|-------------------------------|
| 1. | B. R. Appleton | 27. | G. Ostrouchov |
| 2. | E. F. D'Azevedo | 28-32. | B. W. Peyton |
| 3. | J. B. Drake | 33. | W. M. Post |
| 4. | T. H. Dunigan | 34-38. | S. A. Raby |
| 5. | R. E. Flanery | 39-43. | R. C. Ward |
| 6-10. | G. A. Geist | 44-48. | P. H. Worley |
| 11-12. | R. F. Harbison | 49. | Central Research Library |
| 13-17. | M. T. Heath | 50. | ORNL Patent Office |
| 18. | E. R. Jessup | 51. | K-25 Plant Library |
| 19. | M. R. Leuze | 52. | Y-12 Technical Library |
| 20-24. | F. C. Maienschein | | /Document Reference Station |
| 25. | E. G. Ng | 53. | Laboratory Records - RC |
| 26. | C. E. Oliver | 54-55. | Laboratory Records Department |

EXTERNAL DISTRIBUTION

56. Dr. Loyce M. Adams, Department of Applied Mathematics, University of Washington, Seattle, WA 98195
57. Dr. Christopher R. Anderson, Department of Mathematics, University of California, Los Angeles, CA 90024
58. Dr. Donald M. Austin, 6196 EECS Bldg, University of Minnesota, 200 Union St., S.E., Minneapolis, MN 55455
59. Dr. Robert G. Babb, Department of Computer Science and Engineering, Oregon Graduate Center, 19600 N.W. Walker Road, Beaverton, OR 97006
60. Dr. David H. Bailey, NASA Ames, Mail Stop 258-5, NASA Ames Research Center, Moffet Field, CA 94035
61. Dr. Jesse L. Barlow, Department of Computer Science, Pennsylvania State University, University Park, PA 16802
62. Dr. Edward H. Barsis, Computer Science and Mathematics, P. O. Box 5800, Sandia National Laboratory, Albuquerque, NM 87185
63. Dr. Robert E. Benner, Parallel Processing Division 1413, Sandia National Laboratories, P. O. Box 5800, Albuquerque, NM 87185

64. Dr. Marsha J. Berger, Courant Institute of Mathematical Sciences, 251 Mercer Street, New York, NY 10012
65. Prof. Ake Bjorck, Department of Mathematics, Linkoping University, S-581 83 Linkoping, Sweden
66. Dr. John H. Bolstad, L-16, Lawrence Livermore National Laboratory, P. O. Box 808, Livermore, CA 94550
67. Dr. James C. Browne, Department of Computer Sciences, University of Texas, Austin, TX 78712
68. Dr. Bill L. Buzbee, Scientific Computing Division, National Center for Atmospheric Research, P. O. Box 3000, Boulder, CO 80307
69. Dr. Donald A. Calahan, Department of Electrical and Computer Engineering, University of Michigan, Ann Arbor, MI 48109
70. Mr. Brian M. Carlson, Computer Science Department, Vanderbilt University, Nashville, TN 37235
71. Dr. John Cavallini, Acting Director, Scientific Computing Staff, Applied Mathematical Sciences, Office of Energy Research, U.S. Department of Energy, Washington, DC 20585
72. Dr. Tony Chan, Department of Mathematics, University of California, Los Angeles, 405 Hilgard Avenue, Los Angeles, CA 90024
73. Dr. Jagdish Chandra, Army Research Office, P. O. Box 12211, Research Triangle Park, NC 27709
74. Dr. Melvyn Ciment, National Science Foundation, 1800 G Street N.W., Washington, DC 20550
75. Prof. Tom Coleman, Department of Computer Science, Cornell University, Ithaca, NY 14853
76. Dr. Paul Concus, Mathematics and Computing, Lawrence Berkeley Laboratory, Berkeley, CA 94720
77. Lawrence Cowsar, Department of Mathematics, University of Houston, Houston, TX 77204-3476
78. Dr. Jane K. Cullum, IBM T. J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598
79. Dr. George Cybenko, Center for Supercomputing Research and Development, University of Illinois, 104 South Wright Street, Urbana, IL 61801-2932

80. Ms. Helen Davis, Computer Science Department, Stanford University, Stanford, CA 94305
81. Dr. Yuefan Deng, Applied Mathematics Department, SUNY at Stony Brook, Stony Brook, NY 11794-3600
82. Dr. J. J. Dongarra, 107 Ayres Hall, Department of Computer Science, University of Tennessee, Knoxville, TN 37996-1301
83. Dr. J. J. Dorning (EPMD Advisory Committee) Department of Nuclear Engineering Physics, Thornton Hall, McCormick Rd., University of Virginia, Charlottesville, VA 22901
84. Prof. Larry Dowdy, Computer Science Department, Vanderbilt University, Nashville, TN 37235
85. Dr. Iain Duff, Numerical Analysis Group, Central Computing Department, Atlas Centre, Rutherford Appleton Laboratory, Didcot, Oxon OX11 0QX, England
86. Dr. Stanley Eisenstat, Department of Computer Science, Yale University, P. O. Box 2158 Yale Station, New Haven, CT 06520
87. Dr. Howard C. Elman, Computer Science Department, University of Maryland, College Park, MD 20742
88. Dr. Ian Foster, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439
89. Prof. Geoffrey C. Fox, Department of Physics, Room 229.1, Syracuse University, Syracuse, NY 13244-1130
90. Dr. Chris Fraley, Department of Mathematics and Statistics, Utah State University, Logan, UT 84322-3900
91. Dr. Paul O. Frederickson, NASA Ames Research Center, RIACS, M/S T045-1, Moffet Field, CA 94035
92. Dr. Robert E. Funderlic, Department of Computer Science, North Carolina State University, Raleigh, NC 27650
93. Prof. Dennis B. Gannon, Computer Science Department, Indiana University, Bloomington, IN 47401
94. Dr. W. Morven Gentleman, Division of Electrical Engineering, National Research Council, Building M-50, Room 344, Montreal Road, Ottawa, Ontario, Canada K1A 0R8

95. Dr. Alan George, Vice President, Academic and Provost, Needles Hall, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1
96. Dr. Gene Golub, Computer Science Department, Stanford University, Stanford, CA 94305
97. Dr. Joseph F. Grcar, Division 8331, Sandia National Laboratories, Livermore, CA 94550
98. Dr. William D. Gropp, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439
99. Dr. Eric Grosse, 2C 471, 600 Mountain Avenue, Murray Hill, NJ 07922
100. Prof. John L. Gustafson, Ames Laboratory, 236 Wilhelm Hall, Iowa State University, Ames, IA 50011-3020
101. Prof. Robert M. Haralick (EPMD Advisory Committee) Department of Electrical Engineering, Director, Intelligent Systems Lab, University of Washington, 402 Electrical Engr. Bldg. FT-10, Seattle, WA 98195
102. Dr. Philip J. Hatcher, Department of Computer Science, College of Engineering and Physical Sciences, Kingsbury Hall, Durham, NH 03824
103. Dr. Gerald W. Hedstrom, L-71, Lawrence Livermore National Laboratory, P. O. Box 808, Livermore, CA 94550
104. Dr. Don E. Heller, Physics and Computer Science Department, Shell Development Co., P. O. Box 481, Houston, TX 77001
105. Dr. John L. Hennessy, CIS 208, Stanford University, Stanford, CA 94305
106. Dr. N. J. Higham, Department of Mathematics, University of Manchester, Gtr Manchester, M13 9PL, ENGLAND
107. Dr. Charles J. Holland, Air Force Office of Scientific Research, Building 410, Bolling Air Force Base, Washington, DC 20332
108. Dr. Robert E. Huddleston, Computation Department, Lawrence Livermore National Laboratory, P. O. Box 808, Livermore, CA 94550
109. Dr. Ilse Ipsen, Department of Computer Science, Yale University, P. O. Box 2158 Yale Station, New Haven, CT 06520
110. Dr. Lennart Johnsson, Thinking Machines Inc., 245 First Street, Cambridge, MA 02142-1214
111. Dr. Harry Jordan, Department of Electrical and Computer Engineering, University of Colorado, Boulder, CO 80309

112. Dr. Bo Kagstrom, Institute of Information Processing, University of Umea, 5-901 87 Umea, Sweden
113. Prof. Malvyn Kalos, Cornell Theory Center, Engineering and Theory Center Bldg., Cornell University, Ithaca, NY 14853-3901
114. Dr. Hans Kaper, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439
115. Dr. Alan H. Karp, IBM Scientific Center, 1530 Page Mill Road, Palo Alto, CA 94304
116. Dr. Linda Kaufman, Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974
117. Dr. Robert J. Kee, Applied Mathematics Division 8331, Sandia National Laboratories, Livermore, CA 94550
118. Dr. Kenneth Kennedy, Department of Computer Science, Rice University, P.O. Box 1892, Houston, TX 77001
119. Dr. Thomas Kitchens, ER-7, Applied Mathematical Sciences, Scientific Computing Staff, Office of Energy Research, Office G-437 Germantown, Washington, DC 20585
120. Prof. Clyde P. Kruskal, Department of Computer Science, University of Maryland, College Park, MD 20742
121. Dr. Richard Lau, Office of Naval Research, 1030 E. Green Street, Pasadena, CA 91101
122. Dr. Robert L. Launer, Army Research Office, P. O. Box 12211, Research Triangle Park, NC 27709
123. Dr. Scott A. von Laven, Mission Research Corporation, 1720 Randolph Road, SE, Albuquerque, NM 87106-4245
124. Prof. Tom Leighton, Lab for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139
125. Dr. James E. Leiss (EPMD Advisory Committee) 13013 Chesnut Oak Drive, Gaithersburg, MD 20878
126. Dr. John G. Lewis, Boeing Computer Services, P. O. Box 24346, M/S 7L-21, Seattle, WA 98124-0346
127. Dr. Ted Lewis, Research Director, Oregon Advanced Computing Institute, 19500 SW Gibbs Dr. #101, Beaverton, OR 97006
128. Dr. Heather M. Liddell, Center for Parallel Computing, Department of Computer Science and Statistics, Queen Mary College, University of London, Mile End Road, London E1 4NS, England

129. Dr. Joseph Liu, Department of Computer Science, York University, 4700 Keele Street, Downsview, Ontario, Canada M3J 1P3
130. Dr. Franklin Luk, Electrical Engineering Department, Cornell University, Ithaca, NY 14853
131. Dr. Thomas A. Manteuffel, Department of Mathematics, University of Colorado - Denver, Denver, CO 80202
132. Dr. Anita Mayo, IBM T. J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598
133. Dr. James McGraw, Lawrence Livermore National Laboratory, L-306, P. O. Box 808, Livermore, CA 94550
134. Dr. John Meissen, Oregon Advanced Computing Institute, 19500 SW Gibbs Dr., Suite 110, Beaverton, OR 97006-6907
135. Dr. Paul C. Messina, Mail Code 158-79, California Institute of Technology, 1201 E. California Blvd. Pasadena, CA 91125
136. Dr. Cleve B. Moler, MathWorks, 325 Linfield Place, Menlo Park, CA 94025
137. Dr. Jorge J. More, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439
138. Prof. Neville Moray (EPMD Advisory Committee) Department of Mechanical and Industrial Engineering, University of Illinois, 1206 West Green St., Urbana, IL 61801
139. Dr. Dianne P. O'Leary, Computer Science Department, University of Maryland, College Park, MD 20742
140. Dr. Joseph Oliger, Computer Science Department, Stanford University, Stanford, CA 94305
141. Prof. James M. Ortega, Department of Applied Mathematics, Thornton Hall, University of Virginia, Charlottesville, VA 22901
142. Dr. Peter Pacheco, Mathematics Department, University of San Francisco, San Francisco, CA 94117
143. Prof. Merrell Patrick, Department of Computer Science, Duke University, Durham, NC 27706
144. Dr. James C. Patterson, Boeing Computer Services, P.O. Box 24346, MS 7L-21, Seattle, WA 98124-0346

145. Dr. Peter C. Patton, Patton Associates, Inc., 101 International Plaza, 7900 International Drive, Minneapolis, MN 55425
146. Dr. Linda R. Petzold, L-316, Lawrence Livermore National Laboratory, P. O. Box 808, Livermore, CA 94550
147. Dr. Robert J. Plemmons, Departments of Mathematics and Computer Science, North Carolina State University, Raleigh, NC 27650
148. Dr. Angela Quealy, Sverdrup Technology, Inc., 2001 Aerospace Parkway, Brook Park, OH 44142
149. Prof. Daniel A. Reed, Computer Science Department, University of Illinois, Urbana, IL 61801
150. Dr. John K. Reid, Numerical Analysis Group, Central Computing Department, Atlas Centre, Rutherford Appleton Laboratory, Didcot, Oxon OX11 0QX, England
151. Dr. John R. Rice, Computer Science Department, Purdue University, West Lafayette, IN 47907
152. Dr. Garry Rodrigue, Numerical Mathematics Group, Lawrence Livermore National Laboratory, Livermore, CA 94550
153. Dr. Donald J. Rose, Department of Computer Science, Duke University, Durham, NC 27706
154. Dr. Joel Saltz, ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23665
155. Dr. Ahmed H. Sameh, Computer Science Department, University of Illinois, Urbana, IL 61801
156. Dr. Jorge Sanz, IBM Almaden Research Center, Department K53/802, 650 Harry Road, San Jose, CA 95120
157. Prof. Robert B. Schnabel, Department of Computer Science, University of Colorado at Boulder, ECOT 7-7 Engineering Center, Campus Box 430, Boulder, CO 80309-0430
158. Dr. Robert Schreiber, RIACS, MS 230-5, NASA Ames Research Center, Moffet Field, CA 94035
159. Dr. Martin H. Schultz, Department of Computer Science, Yale University, P. O. Box 2158 Yale Station, New Haven, CT 06520
160. Dr. David S. Scott, Intel Scientific Computers, 15201 N.W. Greenbrier Parkway, Beaverton, OR 97006

161. The Secretary, Department of Computer Science and Statistics, The University of Rhode Island, Kingston, RI 02881
162. Prof. Charles L. Seitz, Department of Computer Science, California Institute of Technology, Pasadena, CA 91125
163. Dr. Andrew Sherman, Department of Computer Science, Yale University, P. O. Box 2158 Yale Station, New Haven, CT 06520
164. Dr. Horst D. Simon, Mail Stop 258-5, NASA Ames Research Center, Moffett Field, CA 94035
165. Dr. William C. Skamarock, 3973 Escuela Court, Boulder, CO 80301
166. Dr. Burton Smith, Tera Computer Company, 400 North 34th Street, Suite 300, Seattle, WA 98103
167. Dr. Marc Snir, IBM T.J. Watson Research Center, Department 420/36-241, P. O. Box 218, Yorktown Heights, NY 10598
168. Prof. Larry Snyder, Department of Computer Science and Engineering, FR-35, University of Washington, Seattle, WA 98195
169. Dr. Danny C. Sorensen, Department of Mathematical Sciences, Rice University, P. O. Box 1892, Houston, TX 77251
170. Dr. Rick Stevens, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439
171. Prof. G. W. Stewart, Computer Science Department, University of Maryland, College Park, MD 20742
172. Mr. Steven Suhr, Computer Science Department, Stanford University, Stanford, CA 94305
173. Dr. Paul N. Swartztrauber, National Center for Atmospheric Research, P. O. Box 3000, Boulder, CO 80307
174. Dr. Wei Pai Tang, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1
175. Dr. Lloyd N. Trefethen, Department of Mathematics, Massachusetts Institute of Technology, Cambridge, MA 02139
176. Dr. Raymond S. Tuminaro, Parallel Processing Division, 1413, Sandia National Laboratories, Albuquerque, NM 87185

177. Prof. Charles Van Loan, Department of Computer Science, Cornell University, Ithaca, NY 14853
178. Dr. Robert G. Voigt, ICASE, MS 132-C, NASA Langley Research Center, Hampton, VA 23665
179. Dr. Michael D. Vose, 107 Ayres Hall, Department of Computer Science, University of Tennessee, Knoxville, TN 37996-1301
180. Mr. Thomas Wagner, Computer Science Department, Vanderbilt University, Nashville, TN 37235
181. Prof. Mary F. Wheeler (EPMD Advisory Committee) Rice University, Department of Mathematical Sciences, P.O. Box 1892, Houston, TX 77251
182. Dr. Andrew B. White, Computing Division, Los Alamos National Laboratory, Los Alamos, NM 87545
183. Office of Assistant Manager for Energy Research and Development, U.S. Department of Energy, Oak Ridge Operations Office, P. O. Box 2001, Oak Ridge, TN 37831-8600
- 184-193. Office of Scientific & Technical Information, P. O. Box 62, Oak Ridge, TN 37831