ORNL/TM--11616

DE91 005648

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

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

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 checking = 0, and enables parameter checking in PICL routines if checking = 1. 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 release = 1.
void load0(char *file, int node)	- loads a program on node number node . if node = -1, 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, open0 allocates numproc node processors, enables interprocessor communication, and returns the host's ID number. On a node, open0 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 msgtype = -1, then probe0 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 bytes (in bytes). If msgtype = -1, then any type is accepted. (<i>Blocking</i>)
void recvinfo0(int *bytes, int *msgtype, int *source)	- returns information about the most recent recv0 or successful probe0 call.
void send0(char *buf, int bytes, int msgtype, int dest)	- sends a message of length bytes (in bytes) to processor number dest . The msgtype 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 who0 , 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

void barrier0()	- executes barrier synchronization of all nodes specified by setarc0 .
void bcast0(char *buf, int bytes, int msgtype, int root)	- broadcasts a message.
void gand0(char *buf, int items, int datatype, int msgtype, int root)	- computes the componentwise "AND" of a distributed set of vectors. Datatypes for logical operations are: 0=char, 1=short, 2=int, 3=long.
void gcomb0(char *buf, int items, int datatype, int msgtype, int root, void (*comb)())	- 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.
void getarc0(int *nprocs, int *top, int *ord, int *dir)	- returns the number of processors and architecture parameters specified by the most recent call to setarc0 .
int ginv0(int i)	- returns the inverse binary reflected Gray code of <i>i</i> .
void gmax0(char *buf, int items, int datatype, int msgtype, int root)	- computes the componentwise maximum of a distributed set of vectors.
void gmin0(char *buf, int items, int datatype, int msgtype, int root)	- computes the componentwise minimum of a distributed set of vectors.
void gor0(char *buf, int items, int datatype, int msgtype, int root)	- computes the componentwise "OR" of a distributed set of vectors.
void gprod0(char *buf, int items, int datatype, int msgtype, int root)	- computes the componentwise product of a distributed set of vectors.
int gray0(int i)	- returns the binary reflected Gray code of <i>i</i> .
void gsum0(char *buf, int items, int datatype, int msgtype, int root)	- computes the componentwise sum of a distributed set of vectors.
void gxor0(char *buf, int items, int datatype, int msgtype, int root)	- computes the componentwise exclusive "OR" of a distributed set of vectors.
void setarc0(int *nprocs, int *top, int *ord, int *dir)	- 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)
- opens a trace file and sets the trace record format. Set format = 0 for use with ParaGraph. Set format = 1 to label trace records with keywords. (<i>Host only</i>)
void traceexit()
- stops tracing.
void traceflush()
- sends local trace information to the trace file.
void tracehost(int tracesize, int flush)
- starts tracing on the host and specifies how large a buffer to reserve for trace information. If flush = 1 , then traceflush is called automatically if the trace buffer fills up. If flush = 0 , then tracing stops if the trace buffer fills up. (<i>Host only</i>)
void traceinfo(int remaining, int event, int compstats, int commstats)
- returns the current tracing parameters, as set by tracelevel , 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)
- 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)
- generates a trace record marking a user-specified event.
void tracemsg(char *message)
- 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)
- starts tracing on a node and specifies how large a buffer to reserve for trace information. If flush = 1 , then traceflush is called automatically if the trace buffer fills up. If flush = 0 , then tracing stops if the trace buffer fills up. If sync = 1 , 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 `tracelhost` 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 `tracelhost`). 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 bcast0( 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 bcast0( 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 precision 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@cornl.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.

END

DATE FILMED

01 / 15 / 91