MARTIN MARIETTA ENERGY SYSTEMS LIBRARIES

3 4456 0315106 9

ORNL/TM-11130

ornl

OAK RIDGE
NATIONAL
LABORATORY

MARTIN MARIETTA

PICL
A Portable Instrumented Communication Library

C REFERENCE MANUAL

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

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

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

Available to DOE and DOE contractors from the Office of Scientific and Technical Information, P.O. Box 62, Oak Ridge, TN 37831; prices available from (615) 576-8401, FTS 626-8401.

Available to the public from the National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Road, Springfield, VA 22161.

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 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-11130

Engineering Physics and Mathematics Division

Mathematical Sciences Section

PICL
A PORTABLE INSTRUMENTED COMMUNICATION LIBRARY
C REFERENCE MANUAL

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

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

Date Published: July 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

MARTIN MARIETTA ENERGY SYSTEMS LIBRARIES

3 4456 0315106 9

Contents

1 Background and overview 1
2 Organization and design 5
  2.1 Low-level primitives 5
  2.2 High-level communication routines 9
  2.3 Execution tracing 11
3 Low-level primitives 15
  `check0` 19
  `clock0` 20
  `close0` 21
  `load0` 23
  `message0` 25
  `open0` 27
  `probe0` 29
  `recv0` 31
  `recvinfo0` 33
  `send0` 35
  `sync0` 37
  `who0` 39
4 High-level communication routines 41
  `barrier0` 47
  `bcast0` 50
  `bcast1` 54
  `gand0, gmax0, gmin0, gor0, gprod0, gsum0, gxor0` 58
  `gcomb0` 63
  `getarc0` 69
  `ginv0` 71
  `gray0` 72
  `setarc0` 73
5 Execution tracing routines 81
  `traceenable` 84
  `traceexit` 86
  `traceflush` 88
  `tracehost` 91
  `traceinfo` 94
  `tracelevel` 96
  `tracemark` 99
  `tracemsg` 101
  `tracenode` 102
6 Trace data format 105
  6.1 Event records 105

6.2 Computation statistics record 109
6.3 Communication statistics record 110
6.4 Trace message record 111
6.5 Compact trace record format 112
6.6 Order of trace records 113
7 Machine dependencies 115
7.1 iPSC/2 117
7.1.1 Legal parameter values 117
7.1.2 Other machine dependencies 118
7.1.3 Performance 119
7.1.4 Compiling and linking 122
7.2 iPSC/860 125
7.2.1 Legal parameter values 125
7.2.2 Other machine dependencies 126
7.2.3 Performance 127
7.2.4 Compiling and linking 130
7.3 Ncube/3200 133
7.3.1 Legal parameter values 133
7.3.2 Other machine dependencies 134
7.3.3 Performance 135
7.3.4 Compiling and linking 138
8 Getting PICL 141
9 Further Information 143
10 References 143
11 Quick Reference List 145

PICL
A PORTABLE INSTRUMENTED COMMUNICATION LIBRARY
C REFERENCE MANUAL

G. A. Geist

M. T. Heath

B. W. Peyton

P. H. Worley

Abstract

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

This report is the PICL reference manual for C programmers. It contains full descriptions of all PICL routines as well as explanations on how to use the routines to write a parallel program. A short users' guide to PICL containing examples of how to use it is available as a separate report.

1. Background and overview

PICL, the software library documented in this report, evolved over a period of years in direct response to user needs. Interest in parallel computation, both for algorithmic research and for practical applications, goes back several years at Oak Ridge National Laboratory (ORNL), which was one of the first three sites to take delivery of the first commercially available hypercube, the Intel iPSC/1. We received a 64-node iPSC/1 in late summer of 1985. Only a few months later, ORNL was one of the first customers to receive an Ncube hypercube, this one also having 64-nodes. Thus, we were presented almost from the beginning with the problem of porting parallel codes across machines with similar architectures, but with incompatible operating systems and message-passing syntax.

The approach we adopted for providing portability between our two hypercubes was to write a common interface library that implemented a generic set of message-passing primitives. The user would write his parallel programs using these generic communication calls, which would in turn invoke the underlying native communication calls on whatever machine was currently being used. Such a capability could have been provided by a set of macros rather than actual subroutines, but we preferred the greater flexibility that subroutines allowed. Our experience since then has confirmed the value of this approach, as we have added capabilities beyond our original intentions. Moreover, our timing tests have shown that the additional subroutine call overhead is negligible compared to the overall communication speed of today's distributed-memory machines.

As we developed an expanding set of algorithms for our hypercubes, certain message-passing patterns occurred repeatedly, such as global broadcast, global summation, and global synchronization. So that users would not have to invent these routines repeatedly, we made these capabilities available in a convenient and portable way by encapsulating them into additional subroutines that in turn called our generic message-passing primitives. Indeed, some homogeneous and regularly structured applications require only these higher-level routines to meet all of their communication needs. With the addition of the high-level routines, the library permitted us to develop new parallel algorithms and applications programs quickly, and in such a way that they could be transported between our two hypercubes with no changes in the source code.

As new message-passing parallel architectures became available, we adapted our programs to them simply by creating new versions of the generic message-passing primitives, but without any changes in the higher-level routines or in user-level source code. Thus, for example, when the second-generation Intel hypercube was released, we were able to get our hypercube programs running on it successfully on the same day our iPSC/2 was delivered, changing nothing but the small underlying library of primitives. Although our initial set of target machines consisted exclusively of hypercubes, the package was easily adapted to message-passing machines based on other interconnection networks, such as the mesh-based Symult (formerly Ametek) 2010, whose programming environment is based on the Cosmic Environment developed at Caltech.

There are limits, however, on the extent to which portability can be provided directly by our approach. In particular, our implementation of the low-level primitives (or at least our programming style in using them) assumes that the underlying message-passing system is interrupt-driven and automatically routes messages between arbitrary pairs of processors. This is in contrast to the synchronous style of message-passing adopted in the Crystalline operating system developed at Caltech, which does not support automatic routing. In consequence, commercial machines such as the Ametek S/14 and the FPS T-Series are not currently included in the range of machines we support.

We have also not implemented our package on any shared-memory or hybrid-memory machines, but there is no reason one could not do so. This might provide a useful alternative for programming such machines in a way that would preserve portability to message-passing architectures. On some pure shared-memory machines, there is a significant performance penalty in this approach, but for non-uniform memory access (NUMA) machines, on which maintaining data locality is an important performance issue (e.g., BBN Butterfly, IBM RP3, Cedar), message passing may ultimately prove to be the most effective programming paradigm.

More recently, we have added substantial new capabilities to our library, again driven by our own needs. In 1988 we began a major new research project on characterizing the performance of parallel algorithms on parallel architectures. An important facet of our approach is to match detailed theoretical performance models to detailed empirical performance data for real parallel applications codes. Thus, we needed a

ready supply of instrumented parallel application codes from which to collect performance data. Since our codes are written in terms of the portable library, this capability was provided simply by instrumenting the library, again without changes to the user-level source code. These new versions of the low-level primitive and high-level communication routines generate execution trace information on demand, with user control of the volume and detail of the data produced. As an added bonus, the tracing capability provides a useful debugging tool as well.

To some extent our motivation in promulgating this package is selfish: any application code written in terms of this library can be used to generate data for our performance characterization project. Thus, it is to our benefit if the library becomes more widely used, both within ORNL and at other institutions. Of course, we also hope that users of the library will benefit from the advantages the package has to offer: portability across message-passing architectures, convenient high-level global communication operations, and optional execution trace data for performance monitoring or debugging. We do not claim to have accomplished these objectives in the most elegant or comprehensive way possible. Our development of the package was entirely driven by the specific needs of individual projects over a significant period of time. We were not trying to provide portability by inventing a new parallel programming paradigm (such as Yale's Linda), nor were we so ambitious as to try to create a whole new operating environment (such as Caltech's Cosmic Environment). We were simply trying to provide users (mainly ourselves) with an efficient and uniform interface for programming a reasonably wide range of message-passing machines, building that interface on top of whatever tools were provided by the native operating systems. We believe we have accomplished those original goals, and along the way we have introduced additional features, such as optional execution tracing, that go beyond the facilities usually provided by machine vendors. We have tried to make reasonable decisions when confronted by the many choices that must be made in designing any software package, but we know that we cannot please all of the people all of the time, so we welcome user feedback, which will be taken into account in future revisions.

2. Organization and design

Currently, we provide a C version of the library to be used by C programs, and we are developing a FORTRAN version to be used by FORTRAN programs. In the meantime, we provide FORTRAN-to-C interface routines so that FORTRAN programs can call the C version of the library. We currently provide full implementations of the library for the Intel iPSC/2, the Intel iPSC/860, and the Ncube/3200 family of hypercube multiprocessors. We also provide subsets of the library for each of the following distributed-memory multiprocessors and multiprocessor programming environments: the Intel iPSC/1, the Cogent, the Symult S2010, the Cosmic Environment, Linda, Unix System V, and the X Window System. The full implementation of the library will be available on all of these target machines and environments in the near future. We also expect this list to grow as new machines and programming environments appropriate for the library appear.

The library is made up of three distinct sets of routines: a set of low-level generic communication and system primitives, a set of high-level global communication routines, and a set of routines for invoking and controlling the execution tracing facility. The rest of this section provides an overview of each of the three components of the package and how they are related to one another. Since several key design issues are discussed here, we strongly recommend that any new user read this section carefully before using the library. The C user should then proceed to the later sections of this document for further information. A short users' guide to PICL containing examples of how to use it is also available as a separate report [Gei90].

A FORTRAN version of the reference manual will be published as a separate report after the development of the FORTRAN version of PICL is finished. In the meantime, FORTRAN users will need to be familiar with both the C version of the documentation and the source code for the FORTRAN-to-C interface routines.

2.1. Low-level primitives

Overview. On each of the target machines, the user has at his disposal a set of machine-dependent system and communication routines that enable him to use the underlying parallel architecture. Because each of the target machines and programming environments supports the same general message-passing programming paradigm,

`open0:`	Open interprocessor communication.
`close0:`	Close interprocessor communication.
`load0:`	Load a node program.
`sync0:`	Synchronize the processors.
`send0:`	Send a message.
`recv0:`	Receive a message.
`probe0:`	Check for an arriving message.
`recvinfo0:`	Return information about the most recently received or probed for message.
`message0:`	Print a message on the standard output device.
`who0:`	Return processor ID number, host ID number, and number of node processors allocated.
`clock0:`	Read the system clock.
`check0:`	Enable or disable parameter checking.

Table 1: Low-Level PICL Routines

most of these system and communication functions are available on all of the machines, though the subroutine names and parameter lists vary from machine to machine. Most importantly, our experience has shown that this set of common functions and capabilities is sufficient to produce efficient, clear, well-written codes. By providing the user with a generic interface to this common pool of functions, these library routines enable the user to write codes that can be transported without change to any machine on which the library has been implemented. The library contains a generic interface routine for each of the functions described in Table 1.

A function or capability available on one target machine but not on another obviously cannot be included in a library intended to to achieve portability for application codes. Thus, we support a restricted programming model that includes only capabilities and functions available on all target multiprocessing environments.

Programming model. Our programming model can be viewed as a generic form of the message-passing programming environments found on most distributed-memory processors. It assumes a set of autonomous processors, each possessing a fixed amount of memory to which no other processor has access. Processors share data by sending messages to each other. More specifically, if processor i has data required by processor

j, then i must send the data to j by issuing a `send0` command, and processor j must issue a `recv0` command in order to receive the message. From the user's viewpoint, processor i is idle (or *blocked*) from the time it issues the `send0` command until the message is copied from the user's message buffer into a system buffer, at which time the user's message buffer can be reused safely. Processor j is 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 the program will never terminate if a `recv0` command is never satisfied by an arriving message. The programmer must design his programs so that there is an incoming message corresponding to each call to `recv0`.

The model also assumes that interprocessor communication is *interrupt-driven*: if processor i calls `send0` to send a message to processor j before processor j calls `recv0` to receive it, then the incoming message causes processor j's operating system to interrupt whatever task it is currently processing in order to receive the message and store it in a system buffer. When the user's program on processor j finally issues a request for the message, the message is copied from the system buffer to the buffer provided by the user's program. Thus, we support an *asynchronous* programming style, rather than a synchronous style where each sending processor blocks until the receiving processor has issued the corresponding `recv0`.

We make no assumptions about the underlying communication network, and we rely on the ability of the target multiprocessors to send messages between arbitrarily chosen pairs of processors. The time required to send a message between two processors will often be 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, the *node* processors (or simply nodes), *only* through the host. Typically, an application code consists of one program that runs on the host and another program (or set of programs) that runs on the node processors. The host program calls system primitives to allocate node processors, load the node program (or programs) on the nodes, send input data required by the node programs, and receive results from the nodes.

Restrictions. While the library implements a somewhat limited programming model, it more than adequately meets our needs, and we have not found excluded features and capabilities to be essential in our programs. The following features, available on some of our target multiprocessors, are not supported:

1) PICL does not support running more than one process (or node program) simultaneously on a single processor.

2) PICL does not support the use of *nonblocking* interprocessor communication.

3) PICL does not support user-defined handlers for system interrupts.

4) PICL does not explicitly support synchronous interprocessor communication.

Although users whose application programs require capabilities not supported by the library will not be able to use this library to achieve full portability, they can still use it to reduce the amount of code they have to write and the number of changes required when porting a program to another machine.

Portability. Our use of generic interface routines that in turn call vendor-supplied native routines raises the following fundamental portability issue. Each of our target multiprocessors has its own range and interpretation of legal parameter values. But, confining the user to universally valid parameter values is too restrictive, and it would become more restrictive as the number of programming environments on which the library is implemented grows. We have chosen therefore to allow the user to use the full range of valid parameter values on each target machine, with the exception of a small subset of values reserved for use by the library itself. On each machine, the routines perform conventional error trapping, checking each input parameter to see if it falls in the range of valid values for that particular machine. When an invalid value is encountered by a routine, an error message is printed on the standard output device of the host and the routine stops processing. Thus, while the software requires the user to select parameter values valid on each of his target machines, when he mistakenly choses a value that causes an error condition on one of the machines, his program fails gracefully, letting him know precisely what happened. To aid the user in selecting portable input parameter values, Section 7 describes the valid range of values of each machine-dependent input parameter on a machine-by-machine basis.

Further information. Section 3 of this document describes in more detail the C version of the low-level communication and system primitives.

2.2. High-level communication routines

The routines included in the high-level communication library are those that have proven most useful in our development of parallel algorithms and application programs for distributed-memory machines. See Table 2 for a description of the routines currently included.

We expect to add other basic high-level communication routines to the library in response to the needs of a growing community of users. On the other hand, we want to maintain the relative simplicity and manageable size of the library, so we will not attempt to provide every conceivable variation and combination of current capabilities. 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 routine(s).

Because we often study how an algorithm designed for message-passing machines behaves on different interconnection topologies, we have designed these routines to run on various network topologies: e.g., hypercube, fully-connected, unidirectional ring, and bidirectional ring interconnection topologies. The user must use the `setarc0` routine to specify which interconnection topology to use. We also expect to add grid topologies as a user option.

The fully-connected option requires further comment. By using automatic routing through intermediate processors, our target machines allow the user to program them as if they were fully connected. Moreover, the "worm-hole" routing now available on the iPSC/2, iPSC/860, and the Symult 2010 enables these machines to approximate the actual behavior of a fully-interconnected network. Thus, despite the fact that a fully-connected network is not physically realized in any of our current target machines, the simulation of such a network is supported in our package.

Section 4 describes in more detail the C high-level communication routines.

`setarc0`:	Specify the underlying topology to be used by the high-level communication routines.
`getarc0`:	Return the parameters used in the most recent call to the `setarc0` routine.
`barrier0`:	Synchronize the processors.
`bcast0`:	Broadcast the elements of a vector from one processor to all other processors.
`bcast1`:	Broadcast the elements of a vector, taking advantage of pipelining during the broadcast.
`gand0`:	Compute the element-wise global "AND" of a distributed set of vectors.
`gcomb0`:	Compute the element-wise global function of a distributed set of vectors where the function is a user-supplied routine.
`gmax0`:	Compute the element-wise global maximum of a distributed set of vectors.
`gmin0`:	Compute the element-wise global minimum of a distributed set of vectors.
`gor0`:	Compute the element-wise global "OR" of a distributed set of vectors.
`gprod0`:	Compute the element-wise global product of a distributed set of vectors.
`gsum0`:	Compute the element-wise global sum of a distributed set of vectors.
`gxor0`:	Compute the element-wise global exclusive "OR" of a distributed set of vectors.
`ginv0`:	Compute the inverse Gray code transformation of an integer.
`gray0`:	Compute the Gray code transformation of an integer.

Table 2: High-Level PICL Routines

`traceenable:`	Enable tracing and specify the name of the trace file.
`tracehost:`	Begin tracing on the host.
`tracenode:`	Begin tracing on a node processor.
`tracelevel:`	Specify the amount and type of trace information.
`traceinfo:`	Return the current `tracelevel` specification.
`tracemark:`	Generate a user-typed trace record.
`tracemsg:`	Write a line of text directly into the trace file.
`traceflush:`	Send trace information to the trace file *now*.
`traceexit:`	Stop tracing.

Table 3: PICL Routines for Tracing

2.3. Execution tracing

When the user requests execution tracing, he activates code within the low-level primitive and/or high-level communication routines that produces time-stamped records detailing the course of the computation on each processor. The information it produces is similar to that produced by some hypercube simulators [Dun86] and can help in analyzing performance or debugging a code. One of the key quantities captured is a record of 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.

It is crucial that the tracing facility have minimal impact on the performance of the code being studied. Three features of our design contribute to that goal. First, the tracing is "event-driven": trace information is generated only when an event of interest has occurred. The default events are sends, receives, and other events connected with interprocessor communication. Thus, tracing is guaranteed to have minimal impact on stretches of computation in which little interprocessor communication occurs.

Second, the user can control the amount and type of trace data that is collected. By using the `tracelevel` command to enable and disable the various types of tracing and to control the level of detail recorded, he can collect detailed data only where needed and avoid generating extraneous data from portions of the code not currently of interest. In the same vein, when his application is sufficiently regular and homogeneous to allow

extensive use of the high-level communication routines, `tracelevel` can be used to trace calls to the high-level routines while simultaneously suppressing trace records for the individual sends and receives that constitute each high-level operation. Another tool that enables removal of unwanted detail is the `tracemark` command. While using `tracelevel` to suppress detailed tracing, the user can post his own "major events" in the trace file with the `tracemark` command. Thus, with prudent use of the `tracelevel` command, the `tracemark` command, and the high-level communication library, the user can use the tracing facility to produce small, yet informative trace files, whose generation has little impact on the performance of his code.

Third, the tracing facility stores trace records into a user-specified block of internal memory. There is no external communication or I/O associated with tracing as long as this internal trace array is not filled to capacity before the computation is complete. Node trace records are then automatically sent back to the host after *all* node programs have completed all other processing.

If a processor's internal trace buffer is filled to capacity before the computation is completed, then the user has several options from which to choose other than simply increasing the buffer size in a subsequent run. First, the user can choose to have the contents of the trace buffer sent to the host automatically when the buffer is full, freeing the buffer to be used to store subsequent trace records. With this choice the user obtains complete execution tracing data, but risks significant changes in the program's behavior due to the cost of sending one or more "full loads" of trace data back to the host while still in the middle of the computation. Second, if the user does not choose this *automatic flushing* option, then no more trace records are saved after the trace buffer is full, and the partial record of the computation is sent back to the host at the end of the program execution. This choice preserves unintrusiveness at the cost of losing some of the trace data. As a third alternative the user can explicitly send trace data back to the host at an appropriate time during the computation. By exercising explicit control, the user can design his code so that sending trace data to the host does not alter the program's behavior in an unacceptable way.

While our approach is inevitably more intrusive than a performance monitor built into the operating system (perhaps with hardware support), such a monitor is not available on most machines and, in any case, would be highly non-portable. Experience

with our trace facility indicates that it generally has a small effect on the performance of the code being instrumented, but not enough to change the important features of its run-time behavior. More detail on the perturbations caused by tracing can be found on a machine-by-machine basis in Section 7.

Section 5 describes in more detail the C routines used to enable and control the collection of trace information. Section 6 describes the format of the trace information.

3. Low-level primitives

Purpose

The low-level primitives are the first and most fundamental of the three components comprising the library. These low-level communication and system interface routines provide a portable syntax for message-passing programs, thereby enabling development of programs that can be run on several distributed-memory multiprocessors with no source code changes. Incorrect usage of the low-level primitives generates an error message and causes the program to terminate.

Machines and Multiprocessing Environments

Implementations of the low-level primitives are available for the following machines and multiprocessing environments:

iPSC/1 hypercube	iPSC/2 hypercube	iPSC/860 hypercube
Ncube/3200 hypercube	Cogent	Symult S2010
Cosmic Environment	Linda	Unix System V
X Window System		

Synopsis

The host and node programs should be linked, respectively, with host and node libraries containing the system and communication primitives listed below. All routines except two are available on both host and node. The routine `load0` is available only on the host, while the routine `sync0` is available only on the node processors.

```
void open0(int *numproc, int *me, int *host)
void close0(int release)
void who0(int *numproc, int *me, int *host)

void send0(char *buf, int bytes, int type, int dest)
void recv0(char *buf, int bytes, int type)
void recvinfo0(int *bytes, int *type, int *source)
int probe0(int type)
void message0(char *message)
```

```
void load0(char *file, int node)

void sync0()

void check0(int checking)
double clock0()
```

Standard Usage

A typical host program calls the following routines (in the indicated order): `open0`, `load0`, `send0`, `recv0`, and `close0`. The call to `open0` allocates a set of processors to the host program and enables interprocessor communication. One or more calls to `load0` load node programs on the node processors. One or more calls to `send0` send data to the nodes. One or more calls to `recv0` receive results back from the nodes. The call to `close0` disables interprocessor communication.

A typical node program minimally calls the following routines (in the indicated order): `open0`, `recv0`, `send0`, and `close0`. The call to `open0` to the enables interprocessor communication. One or more calls to `recv0` receive data from the host. One or more calls to `send0` send results back to the host. Communication with other node processors will require additional `send0` and `recv0` calls. The call to `close0` disables interprocessor communication *permanently*.

The next two pages contain node and host programs that demonstrate how the low-level PICL routines are typically used. These programs take a vector of floating point numbers and calculate a new vector that is the generated by summing consecutive pairs of elements in the original vector. For example, if the two vectors are called `old` and `new`, then `new`$[i]$ = `old`$[i]$ + `old`$[i+1]$. Although many of the PICL routines are not used, all of the required ones are called.

```
main(){
  int P, me, host, M, N, j, k, bytes, type, node;
  float *data, *results, *block;
  double btime, etime, clock0();

  /* start timing */
  btime = clock0();

  /* allocate processors and open communication channel */
  P = 32;
  open0(&P, &me, &host);

  /* allocate space for data and results, and create data */
  N = 1024;
  M = N/P;
  data = (float *)malloc(N*sizeof(float));
  results = (float *)malloc(N*sizeof(float));
  block = (float *)malloc(M*sizeof(float));
  for (j=0; j<N; j++) data[j] = 1.0;

  /* load node program */
  load0(``nodeprogram'', -1);

  /* send data to processors */
  for (j=0; j<P; j++) send0(&M, sizeof(int), 0, j)
  for (j=0; j<P; j++) send0(&data[P*j], M*sizeof(float), 1, j)

  /* receive results */
  for (j=0; j<P; j++){
    recv0(block, M*sizeof(float), 3);
    recvinfo0(&bytes, &type, &node);
    for (k=0; k<M; k++) results[M*node + k] = block[k];
    };

  /* finish timing and send message to the standard output */
  etime = clock0();
  printf("host took %lf seconds to finish", etime-btime);

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

Figure 1: Example C host program using low-level PICL routines.

```
main(){
  int P, me, host, M, left, j;
  float *data, *results, rightnum;
  double btime, etime, clock0();
  char buf[80];

  /* open communication channel */
  open0(&P, &me, &host);

  /* receive problem parameter from host */
  recv0(&M, sizeof(int), 0);

  /* allocate space for data and results */
  data = (float *)malloc(M*sizeof(float));
  results = (float *)malloc(M*sizeof(float));

  /* receive problem data from host */
  recv0(data, M*sizeof(float), 1);

  /* synchronize and begin timing the computational kernel*/
  sync0();
  btime= clock0();

  /* communicate with other processors and add  */
  /* successive numbers together */
  left = (me -1) % P;
  send0(&data[0], sizeof(float), 2, left);
  for (j=0; j<M-1; j++) results[j] = data[j] + data[j+1];
  recv0(&rightnum, sizeof(float), 2);
  results[M-1] = data[j-1] + rightnum;

  /* finish timing and send result to standard output */
  etime = clock0();
  sync0();
  sprintf(buf, "node %d took %lf seconds to finish", me, etime-btime);
  message0(buf);

  /* send results to host and close communication channel */
  send0(results, M*sizeof(float), 3, host);
  close0();
}
```

Figure 2: Example C node program using low-level PICL routines.

check0

`check0` *enables or disables parameter checking in the low-level primitives.*

Environment

host, node

Synopsis

```
void check0(checking)
int checking;
```

Input parameters

`checking` – If `checking` is 1, then parameter checking is enabled. Otherwise, it is disabled.

Output parameters

None

Discussion

Often a parameter passed to a PICL routine is simply passed directly to a machine-specific system primitive. Since the range of valid values for such a parameter often varies from machine to machine, this raises an important portability issue. When parameter checking is enabled, each PICL routine examines such parameters to see if they are valid on the current multiprocessor. When an invalid value is discovered, the PICL routine generates an error message and stops execution.

Since by default parameter checking is enabled, a call to `check0` is necessary to disable parameter checking. When parameter checking is disabled, many of the PICL routines run slightly faster. But parameter checking should be disabled only for programs that are known to be correct.

clock0

`clock0` *returns the system clock time (in seconds).*

Environment

host, node

Synopsis

```
double clock0();
```

Input parameters

None

Output parameters

None

Function value

`clock0` returns as a double precision value the system clock time converted into seconds.

Discussion

The `clock0` function first converts the value obtained from the native timing routine to seconds, and then returns this value as a double precision number. Unlike most of the low-level primitives, `clock0` performs correctly even if interprocessor communication has not been enabled with a call to `open0` or has been disabled with a call to `close0`.

close0

`close0` *disables interprocessor communication. The host version of* `close0` *can release the current allocation of node processors on machines that have a system primitive for that purpose.*

Environment

host, node

Synopsis

```
void close0(release)
int release;
```

Input parameters

`release` – On the host of a machine that can deallocate node processors under program control, if `release` is 1, then `close0` releases the node processors allocated to the user. If `release` contains a value other than 1, then `close0` releases no allocated node processors. On a node processor, the parameter is ignored.

Output parameters

None

Discussion

Every processor, including the host, must execute `open0` before it can execute any low-level primitive that performs, or even indirectly supports, interprocessor communication. The user can also explicitly disable interprocessor communication with a call to `close0`. When interprocessor communication is disabled, calling the routines `close0`, `load0`, `probe0`, `recv0`, `recvinfo0`, `send0`, or `who0` will generate an error message stating that the communication channel is not open, and the program will terminate. As a general rule, a user should issue

a call to `open0` near the beginning of his program (host or node) and a call to `close0` near the end.

To guarantee correct execution, any node program that uses PICL routines must call `close0`. On the nodes, it is best to assume that `close0` also kills the process since this is how it is implemented on some machines. Thus, `close0` should be the last executable instruction in a node program.

Similarly, if the host program uses PICL routines, then it must call `close0` in order to guarantee correct execution. But, it is logically correct to call `open0` and `close0` multiple times on the host as long as a call to `close0` is used to disable communication before the next call to `open0` or before the end of the program. If `release` is equal to one, then `close0` releases the allocated processors in addition to disabling interprocessor communication, provided that the multiprocessor is capable of doing this under program control. On machines incapable of releasing processors while under program control, the value of `release` is ignored. The only practical issue raised by releasing processors in this way is the correct choice of parameters in a subsequent call to `open0`. The manual page for `open0` discusses this in detail.

load0

`load0` *loads and starts the execution of an executable file on one node processor or on every node processor allocated to the user.*

Environment

host

Synopsis

```
void load0(file, node)
char *file;
int node;
```

Input parameters

`file` – `file` points to the name of the file containing the executable to be loaded and run.

`node` – `node` is the ID number of the processor on which the executable is to be loaded and run. When its value is -1, the executable is loaded on every allocated node processor.

Output parameters

None

Discussion

The `load0` routine is available only on the host. It is the only means provided by the package for loading and running node programs. A call to `load0` will fail if interprocessor communication has not been enabled with a call to `open0`.

If P node processors have been allocated, then the legal values for the input parameter `node` are the integers between -1 and $P-1$. The integers 0 to $P-1$ are the ID numbers of the individual node processors, and are used to load a node program on a single processor. Calling `load0` with `node` set to -1 loads the same

program on every allocated node processor. If different programs are to run on different node processors, then the user must load each processor individually.

PICL does not support having multiple processes on a single processor. If `load0` is used to load a process onto a processor that already has an active process, then the result is machine-dependent. On some multiprocessors, the original process is killed and the new process begins execution normally. On other multiprocessors, an error message is generated and execution stops. See Section 7 for a description of the behavior on a given target multiprocessor. Note that it is always legal to load a process onto a processor if a process previously loaded onto the processor has successfully terminated (after calling `close0`), but it is up to the user to determine whether or when this has occurred.

message0

message0 *writes a string on the host's standard output device.*

Environment

host, node

Synopsis

```
void message0(string)
char *string;
```

Input parameters

`string` – `string` points to a character string that is to be printed on the standard output device. Only the first 80 characters will be printed.

Output parameters

None

Discussion

The **message0** routine is intended to be an alternative to the C routine `printf`. Typically, a user will first use the C routine `sprintf` to fill a character buffer. He will then use **message0** to send the buffer to the host, where its contents are printed on the standard output device. Thus, **message0** provides the same functionality as `printf`, although it is limited to relatively small messages ($\leq$ 80 characters).

The primary reason for its inclusion in the library is that the capability of writing to the standard output from a node program is not supported on some target multiprocessors. Also, our experience indicates that using **message0** in a node program is significantly less expensive than using `printf` on those machines that do support `printf` on the nodes. A call to **message0** in a node program will fail

if interprocessor communication has not been enabled with a call to open0. This restriction does not hold for a host program.

open0

`open0` *enables interprocessor communication. The host version of* `open0` *can also allocate processors on machines that provide a system routine for that purpose.*

Environment

host, node

Synopsis

```
void open0(numproc, me, host)
int *numproc, *me, *host;
```

Input parameters

`numproc` – On a node processor, `numproc` is exclusively an output parameter. On the host, if `numproc` points to a positive value, then `open0` will allocate that many processors. If, on the other hand, `numproc` points to a non-positive value, then the interpretation of `numproc` is machine-dependent. For futher discussion of how non-positive values are interpreted read the discussion section below.

Output parameters

`numproc` – `numproc` points to the number of processors allocated.

`me` – `me` points to the ID number of the processor on which the program is running.

`host` – `numproc` points to the ID number of the host processor.

Discussion

On both host and node, interprocessor communication must be "opened" with a call to `open0` before any of the following PICL routines can be called: `close0`, `load0`, `probe0`, `recv0`, `recvinfo0`, `send0`, `sync0`, and `who0`. For the nodes only,

this is also true for `message0`. As a general rule, the user should issue a call to `open0` near the beginning of his program (host or node) and a call to `close0` near the end.

On the host, `open0` is also used to allocate processors on which to load node programs. If `numprocs` points to a positive value, then the routine attempts to allocate that many processors. If the requested number of processors is unavailable, then the underlying primitive prints an error message and stops processing. On hypercubes, the number of processors allocated is required to be a power of two, but PICL essentially frees the user from this restriction. For example, a host program using `open0` to request 13 processors on a hypercube multiprocessor will cause the system to allocate 16 processors. Nevertheless, PICL allows the program access to only the first 13 processors of the allocation, and, upon exit from `open0`, `numprocs` points to the value 13, *not* 16. For all practical purposes, both the host and node programs operate as if the extra three processors were not included in the original allocation.

When `numprocs` points to a nonpositive value, the action taken by `open0` on the host is machine-dependent. On machines where processors can be allocated at the keyboard with a system command (e.g., the iPSC/1, iPSC/2, and iPSC/860), a nonpositive value tells the host program to use the current allocation of nodes. On machines that allow processor allocation only under program control on the host, a nonpositive value tells `open0` to allocate as many processors as possible.

On the host, interprocessor communication can alternate between the "open" and "closed" state as desired by alternating calls to `open0` and `close0`. On machines that allow `open0` to reuse the previous allocation of nodes, a call to `open0` that presumes an existing allocation of nodes must not follow a call to `close0` that releases the current allocation. For example, on the iPSC/2 a call of the form `close0(1)` releases any allocated processors. In the next call to `open0`, `numprocs` must point to a positive value, and thus allocate a new set of nodes. Otherwise, an error message will be generated and the program will stop. Similarly, on such a machine, a call of the form `close0(0)` must be followed by a call to `open0` where `numprocs` points to a nonpositive value in order to reuse the same allocation of processors.

probe0

`probe0` *checks whether a message of specified type is waiting in the message queue. It is nonblocking.*

Environment

host, node

Synopsis

```
int probe0(type)
int type;
```

Input parameters

`type` – `type` contains the type field of the message being sought. When its value is -1, any message in the message queue will satisfy the probe.

Output parameters

None

Function value

`probe0` returns the value 1 if a message of the specified type is found in the message queue. Otherwise, it returns the value 0.

Discussion

Every message has associated with it an integer "type" field that is used by the receiving processor to discriminate one kind of message from another. The `probe0` routine searches the message queue for messages of a specified type. It returns as soon as it has finished checking the queue, whether the desired message has been found or not. If the specified type is -1, then any message will satisfy the probe. A call to `probe0` will fail if interprocessor communication has not been

enabled with a call to **open0**. The routine **recvinfo0** can be used to obtain the following information about a message found by **probe0**: the length and type of the message and the ID number of the originating processor.

Acceptable values for the input parameter **type** are machine-dependent. The range of valid values for this parameter is given on a machine-by-machine basis in Section 7.

The inclusion of **probe0** in the package is essential to support the asynchronous programming style necessary for efficiency in some applications. Typically, such an application program will periodically probe the message queue for the next message that is expected. A blocking probe, also useful in some applications, can be coded as follows:

```
while (probe0(type) == 0) ;
```

recv0

`recv0` *receives a message of the specified type. It blocks processing until the message is received.*

Environment

host, node

Synopsis

```
void recv0(buf, bytes, type)
char *buf;
int bytes, type;
```

Input parameters

`buf`	–	`buf` points to the beginning of the buffer into which the incoming message is to be written.
`bytes`	–	`bytes` is the size of the buffer (in bytes) into which the message is to be written.
`type`	–	`type` is the "type" of the message to be received. When its value is -1, `recv0` will receive the first available message of any type.

Output parameters

None

Discussion

The routine `recv0` receives a message of type `type` into a buffer `buf` of length `bytes`. If the value of `type` is -1, then the first available message of any type will be received. When executing `recv0`, the processor stops all computation until a message satisfying the request is received. A call to `recv0` will fail if interprocessor communication has not been enabled with a call to `open0`. The routine `recvinfo0` can be used to obtain the following information about the most

recently received (or successfully probed for) message: the length and "type" of the message and the ID number of the originating processor.

Acceptable values for the input parameter **type** are machine-dependent. The range of valid values for this parameter is given on a machine-by-machine basis in Section 7.

recvinfo0

`recvinfo0` *returns information about the most recently received or successfully probed for message.*

Environment

host, node

Synopsis

```
void recvinfo0(bytes, type, source)
int *bytes, *type, *source;
```

Input parameters

None

Output parameters

`bytes`	–	`bytes` points to the length (in bytes) of the most recently received or successfully probed for message.
`type`	–	`type` points to the "type" of the most recently received or successfully probed for message.
`source`	–	`source` points to the processor ID number of the originating processor of the most recently received or successfully probed for message.

Discussion

The routine `recvinfo0` reports information about the most recently received or successfully probed for message. A call to `recvinfo0` will fail if interprocessor communication has not been enabled with a call to `open0`.

If P node processors have been allocated, then the possible values for `source` are the integers between 0 and $P-1$, the ID numbers of the P node processors, and the integer $-32,768$, the ID number of the host processor.

If a program mixes calls to the native receive routine with calls to `recv()`, then the information `recvinfo()` returns may or may not pertain to the most recently received message. This problem will not arise if all incoming messages are handled by the `probe()` and `recv()` routines.

send0

send0 *sends a message. It blocks processing until it is safe to modify the contents of the send buffer.*

Environment

host, node

Synopsis

```
void send0(buf, bytes, type, dest)
char *buf;
int bytes, type, dest;
```

Input parameters

buf	–	buf points to the beginning of the buffer containing the message to be sent.
bytes	–	bytes is the length of the message (in bytes).
type	–	type allows the user's program to distinguish between different kinds or "types" of messages .
dest	–	dest is the processor ID number of the destination processor.

Output parameters

None

Discussion

The routine send0 sends a message stored in the buffer pointed to by buf to the processor whose ID number is contained in dest. The length and "type" of the message are contained in bytes and type respectively. A call to send0 will fail if interprocessor communication has not been enabled with a call to open0. Note that the host ID number is $-32,768$, but this value is also returned by open0 and who0 and need not be explicitly used in a program.

The send operation invoked by **send0** is both blocking and asynchronous. That is, computation on the sending processor resumes as soon as the message is safely on its way to the receiving processor. This is in contrast to synchronous communication, during which computation on the sending processor halts until the matching receive is executed by the receiving processor. It is also in contrast to nonblocking communication, in which the sending processor resumes computation before the message has been copied from the user's message buffer.

Acceptable values for the input parameters **bytes**, **type**, and **dest** are machine-dependent. The range of valid values for these two parameters is given on a machine-by-machine basis in Section 7.

sync0

`sync0` *synchronizes the node processors.*

Environment

node

Synopsis

```
void sync0()
```

Input parameters

None

Output parameters

None

Discussion

One way to (approximately) synchronize the node processors is to call `sync0` on each node processor. When a node processor executes `sync0`, it participates in an interchange of messages, the sole purpose of which is the nearly simultaneous exit of all processors from `sync0`. These message exchanges are designed to minimize as much as possible the time that elapses between the first and last exit from `sync0`.

A call to `sync0` will fail if interprocessor communication has not been enabled with a call to `open0`. A call to `sync0` is also likely to fail if it is immediately preceded by a call to `recv0` with a `type` parameter of -1. In this situation, the "promiscuous" receive is likely to remove one of the synchronization messages from the queue, after which the synchronization process will never complete because of the missing message. While the library is able to identify this error and issue an appropriate error message, it is not able to recover from it. Finally, a call to `sync0` will fail unless *all* node processors call `sync0`. If any node processor fails to call `sync0`, then all of the processors that do call `sync0` will never exit the routine.

The sync0 routine is very different from the other low-level primitives included in the library. It does not merely issue a call to some native routine whose function clearly matches its own. It is more like the high-level communication routines than the low-level primitives with which it is grouped. Despite the fact that vendors do not generally supply a convenient barrier or synchronization routine, we feel that this function is of such fundamental importance in obtaining legitimate timing results that we include it in the low-level library.

who0

who0 returns the number of processors, the node ID number and the host ID number.

Environment

host, node

Synopsis

```
void who0(numproc, me, host)
int *numproc, *me, *host;
```

Input parameters

None

Output parameters

numproc	–	numproc points to the number of processors allocated to the user.
me	–	me points to the ID number of the processor on which the program is running.
host	–	host points to the ID number of the host processor.

Discussion

The who0 routine returns the same information that open0 returns. This routine is included so that this information can be conveniently obtained by subroutines that do not call open0. A call to who0 will fail if interprocessor communication has not been enabled with a call to open0.

4. High-level communication routines

Purpose

The high-level communication routines perform various operations that involve global communication, such as global broadcast, global summation, and global synchronization. Since the only routines called by the high-level routines are low-level PICL primitives, the source code for the high-level routines is independent of the target machine.

While we include only those global communication routines that we have found useful to date in *our* codes, these routines are also meant to serve as models for routines that implement similar global communication operations not currently implemented in our package. *Note that the high-level communication routines do not check for errors (beyond that provided by the low-level PICL primitives), and some extra care must be taken to use the routines correctly.*

Machines and Multiprocessing Environments

Implementations of the high-level communication routines are available for the following machines and multiprocessing environments:

iPSC/1 hypercube	iPSC/2 hypercube	iPSC/860 hypercube
Ncube/3200 hypercube	Cogent	Symult S2010
Cosmic Environment	Linda	Unix System V
X Window System		

Synopsis

The host and node programs should be linked, respectively, with host and node libraries containing the high-level communication routines listed below. With the exception of `barrier0`, every routine is available on both host and node. The routine `barrier0` is available only on the nodes.

```
void setarc0(int *nprocs, int *top, int *ord, int *dir)
void getarc0(int *nprocs, int *top, int *ord, int *dir)
```

```
void bcast0(char *buf, int bytes, int type, int root)
void bcast1(char *buf, int bytes, int type, int root)

void gcomb0(char *buf, int items, int datatype,
            int msgtype, int root, void (*comb)())

void gand0(char *buf, int items, int datatype,
           int msgtype, int root)
void gmax0(char *buf, int items, int datatype,
           int msgtype, int root)
void gmin0(char *buf, int items, int datatype,
           int msgtype, int root)
void gor0(char *buf, int items, int datatype,
          int msgtype, int root)
void gprod0(char *buf, int items, int datatype,
            int msgtype, int root)
void gsum0(char *buf, int items, int datatype,
           int msgtype, int root)
void gxor0(char *buf, int items, int datatype,
           int msgtype, int root)

void barrier0()

int gray0(int i)
int ginv0(int i)
```

Standard Usage

The high-level communication routines supplied with PICL were designed to serve a number of purposes:

- as a convenience to users, so that certain commonly occurring global communication operations do not have to be reimplemented by each programmer,
- as examples of machine-independent code written using PICL primitives, serving as models upon which users can base other global operations of their own design,

- as abstractions above the low-level communication primitives, reducing the volume of trace data produced by PICL's instrumentation, and
- as a mechanism for conveniently carrying out computational experiments in which the user varies such parameters as the number of processors and the effective interconnection topology used.

The high-level routines are based on the concept of a virtual architecture consisting of a specified number of processors and a specified network topology interconnecting them. The number of processors used can be any number up to the number allocated by the most recent call to `open0`. If the specified interconnection network is not physically realized on a particular machine, then the network is emulated by system-supplied message routing. The architectural parameters are set by calling the routine `setarc0`. The parameters to `setarc0` are input parameters on the host and output parameters on the nodes. The host version of `setarc0` communicates this information to the nodes by sending a message of a special type to each of the nodes to be used. In this way, the host program informs the nodes of the number of processors to be used and the interconnection pattern to use for global communication operations. We will refer to the number of processors specified in the call to `open0` as the number of processors *allocated*, and to the number of processors specified in the call to `setarc0` as the number of processors *in use*. Any allocated processors that are not *in use* are unaffected by calls to the high-level routines (other than `setarc0` itself; see below).

For experimenting with varying numbers of processors within a single run, the "standard usage" for the high-level routines is for node programs to contain an outer loop around a call to `setarc0`. All of the processors "hang" in a call to `recv0` inside `setarc0`, awaiting the special message from the host informing them of the number of processors in use, as well as the other architectural parameters. Those processors that have been allocated but are not to be used for the moment will simply remain hung in `setarc0` until a new call to `setarc0` on the host specifies a large enough number of processors to involve them. In order to break out of this loop, a special case is provided; namely, if the number of processors is set to 0 in the call to `setarc0` on the host, then all of the allocated nodes calling `setarc0` will receive a message reporting this fact. In this case, the call

to `setarc0` on the nodes will return the value 0 for the number of processors, so that this condition can be used in a test to break out of the loop and proceed with an orderly termination of the program (including a call to `close0`).

The next two pages contain a sample skeleton code representing a hypothetical application using the high-level routines. The example illustrates how the node program can be executed multiple times, using a different number of processors each time. The node program calls `setarc0` inside an infinite `while` loop that is terminated when `setarc0` returns a value of zero for the number of *in use* processors. The use of varying numbers of processors would be typical of a computational experiment to determine the speedup curve for a parallel program on a given application problem. The other architectural parameters can also easily be varied within this same framework.

```
main()
{
  int nprocs, me, host, p, top, ord, dir, n;
  float *results;

  n = 100;
  results = (float *)malloc0(n*sizeof(float));

  nprocs = 64;
  open0(&nprocs, &me, &host);
  load0("node", -1);
  top = 1;
  ord = 1;
  dir = 1;
  for (p=1; p<=nprocs; p*=2){
    /* set architectural parameters */
    setarc0(&p, &top, &ord, &dir);

    /* broadcast problem data to all processors in use */
    bcast0(&n, sizeof(int), 0, host);

    /* collect global sum of results from all processors in use */
    gsum0(results, n, 4, 1, host);
    }

  p = 0;
  setarc0(&p, &top, &ord, &dir);
  close0(1);
}
```

Figure 3: Example C host program using high-level PICL routines.

```
main()
{
  int p, me, host, top, ord, dir, n;
  float *results;

  open0(&p, &me, &host);
  while (1) {
    /* get architectural parameters from host */
    setarc0(&p, &top, &ord, &dir);
    if (p == 0) break;

    /* get problem data from host */
    bcast0(&n, sizeof(int), 0, host);

    results = (float *)malloc0(n*sizeof(float));
    .
    .
    /* compute local contribution to results vector */
    .
    .
    /* globally sum results across processors and send to host */
    gsum0(results, n, 4, 1, host);

    free(results);
    }

  close0();
}
```

Figure 4: Fragment of C node program using high-level PICL routines.

barrier0

`barrier0` *synchronizes the node processors currently in use.*

Environment

node

Synopsis

```
void barrier0()
```

Input parameters

None

Output parameters

None

Discussion

The routine `barrier0` synchronizes the node processors declared *in use* during the most recent call to `setarc0`. To perform the synchronization, the user should have each node processor *in use* call `barrier0` at the point in the program requiring synchronization. While in `barrier0`, each processor participates in a sequence of message exchanges, the sole purpose of which is the nearly simultaneous exit of all processors from `barrier0`. More precisely, the message exchanges are designed to minimize as much as possible the time that elapses between the first and last escape from `barrier0`. The message types used in the barrier algorithm are determined from a machine-dependent base value and from the number of processors specified by the most recent call to `setarc0`. For example, if `basetype` is the base value and `nprocs` is the number of processors specified by setarc0, then all message types used by `barrier0` are between `basetype` and `basetype` + `nprocs`. This range is guaranteed to be within the legal range of message types for the given multiprocessor. The base value used for a given target machines is described in Section 7.

A call to `barrier0` must be preceded by a call to `open0` to open interprocessor communication, and by a call to `setarc0` to declare the number of processors *in use* and to indicate what interconnection topology to use. For more details consult the manual pages for these two routines.

Like `barrier0`, the low-level primitive routine `sync0` is also designed to sychronize processors, and the two routines are very similar. They differ however in two important ways:

1. Like all of the high-level routines, the source code for `barrier0` is machine-independent; it uses the same hypercube-style dimensional exchange algorithm on all target machines. Since the primary emphasis in `sync0` is on speed of execution and sharpness of the resulting synchronization, the implementation of `sync0` takes advantage of the most efficient synchronization mechanism offered by a given architecture.
2. `barrier0` synchronizes only the processors currently *in use*, as specified in the most recent call to `setarc0`; `sync0` synchronizes all processors in the current allocation of node processors, as specified in the call to `open0`;

The dimensional exchange algorithm used by `barrier0` is likely to fail if one or more of the calls to `barrier0` is immediately preceded by a call to `recv0` with a `type` parameter of -1. In this situation, at least one of these "promiscuous" receives is likely to receive a synchronization message, causing at least some of the processors *in use* to "hang" in `barrier0` due to missing messages. Not only is the routine unable to recover from this error, it is not even able to identify it and issue an appropriate error message. Thus, it is important that the user's program never allow "promiscuous" receives to interfere with `barrier0` in this way.

During the execution of `barrier0`, it is also important that the message queue of each node processor not contain a "non-synchronization" message with a `type` parameter that falls in the range used by `barrier0`. In this case `barrier0` will not deadlock, but the program will not execute correctly from this point on. The user can ensure that this does not occur by using only message types that are smaller than `basetype` (as described in Section 7 for each target machine). Since

`basetype` is near the high end of the range of legal message types, portability can be enhanced by keeping user-defined message types smaller than `basetype` for all machines on which the program is likely to be executed.

bcast0

`bcast0` *broadcasts a vector from one processor to every node processor currently in use. The broadcast is monolithic in that each processor receives the message and completes all forwarding of it before the processor resumes computation. In particular, the processors do not to interrupt the broadcast to perform computations with incoming broadcast data before forwarding the data.*

Environment

host, node

Synopsis

```
void bcast0(buf, bytes, type, root)
char *buf;
int bytes, type, root;
```

Input parameters

`buf` – On the `root` processor, `buf` points to a buffer containing the message to be broadcast. On processors other than the `root`, `buf` points to the buffer where the broadcast message is to be stored.

`bytes` – On the `root` processor, `bytes` is the length (in bytes) of the message to be broadcast. On processors other than the `root`, `bytes` is the length (in bytes) of the buffer in which the broadcast message is to be placed.

`type` – `type` is the message type used when sending and receiving the individual messages that constitute the broadcast.

`root` – `root` is the processor ID number of the source of the broadcast message. Note that `root` can be the host.

Output parameters

None

Discussion

This routine broadcasts data from one processor, designated as the `root`, to every node processor declared *in use* by the most recent call to `setarc0`. Note that the host or any node processor currently *in use* can be the `root`. To perform a global broadcast from a node processor, each node processor *in use* must call `bcast0` with the appropriate input parameters. As the processors enter `bcast0`, they cooperate in moving the data from the `root` to the other node processors *in use*, using the interconnection topology set by the most recent call to `setarc0`. After the last processor has exited `bcast0`, every node processor *in use* contains the broadcast data in the buffer pointed to by `buf`.

When a node processor is the `root`, the host does not participate in any way in the broadcast. Indeed, the host can never receive broadcast data; only node processors *in use* can receive broadcast data. Thus, it never makes sense to call `bcast0` on the host with `root` set to a node processor ID number. We have nevertheless found it convenient to allow the host to be the `root` of a broadcast. To broadcast from the host, the host and every node processor *in use* must call `bcast0` with the parameter `root` set to the host ID number. When this is done, the host first sends the broadcast data to node 0, then the broadcast proceeds just as if it were initially `rooted` at node 0. (Note that the host ID number is a large negative number that can be obtained from `open0` or `who0`. For more details consult the manual pages for these routines.)

A call to `bcast0` must be preceded by a call to `open0` to open interprocessor communication, and by a call to `setarc0` to declare the number of node processors *in use* and to indicate what interconnection topology to use. For details on how the various architecture options affect the way the broadcast is performed, consult the manual page for `setarc0`.

`bcast0` performs a "synchronous" broadcast in the sense that it does *not* allow processors to perform computations with incoming broadcast data before their participation in the broadcast is complete. In contrast, the routine `bcast1` performs an "asynchronous" broadcast designed to allow each processor to perform computation with incoming broadcast data before resuming participation in the

broadcast. Some algorithms use pipelining techniques that require immediate processing of incoming broadcast data. These pipelining techniques can be easily implemented using `bcast1`. For more details, consult the manual page for `bcast1`.

Valid values for the input parameters `bytes` and `type` are machine-dependent. Though the routine `bcast0` performs no machine-dependent parameter checking, these two parameters are routinely checked for validity in the low-level communication routines `recv0` and `send0` to which they are passed unchanged. For each target machine, the range of valid values for each of these two parameters is given in Section 7.

As with most of the high-level communication routines, certain rules for "consistency" among the input parameters must be followed to ensure successful completion of the operation. First, the user must make sure that each non-`root` processor provides a buffer large enough to hold the broadcast message. In particular, `bytes` on the `root` processor must not exceed `bytes` on any non-`root` processor *in use*. If this condition is not met, then at least one processor will fail inside the native receive routine because there is insufficient buffer space to handle the incoming broadcast message (or on some systems the message is simply truncated if the buffer provided is too small). Second, since the input parameter `type` is used to identify the individual messages that constitute a particular broadcast, each processor must call `bcast0` with the same `type` parameter. Failure to do so is likely to result in one or more processors "hanging" inside `bcast0` because an expected incoming broadcast message is missing.

During the execution of `bcast0`, it is important that the message queue on each processor *in use* not contain a "non-broadcast" message with the same message type as that used by the `bcast0`. If `bcast0` happens to receive such a non-broadcast message, then the results are unpredictable. The simplest way to avoid this problem is to avoid "reusing" the broadcast `type` value as a message type anywhere else in the code. It is also important that a broadcast message generated by a call to `bcast0` be removed from the message queue only by a call to `bcast0`. Thus it is dangerous when a call to `bcast0` is immediately preceded by a call to `recv0` with a `type` parameter whose value is -1. In this situation,

the "promiscuous" receive is likely to receive the broadcast message, causing the processor to "hang" in `bcast0` due to the missing message. Not only is the routine unable to recover from this error, it is not even able to identify it and issue an appropriate error message.

bcast1

Used in conjuction with `recv0`, `bcast1` *broadcasts a vector from one processor to every node processor currently in use. In* `bcast1` *the reception and forwarding of the message are decoupled in order to permit the processors to perform computations with incoming broadcast data before the processor's participation in the broadcast is complete.*

Environment

host, node

Synopsis

```
void bcast1(buf, bytes, type, root)
char *buf;
int bytes, type, root;
```

Input parameters

`buf` – On the `root` processor, `buf` points to a buffer containing the message to be broadcast. On processors other than the `root`, `buf` points to a buffer containing the message previously received (with `recv0`), and which is to be forwarded to subsequent processors in the broadcast.

`bytes` – On the `root` processor *only*, `bytes` is the length (in bytes) of the message to be broadcast. On processors other than the `root`, `bytes` is the length (in bytes) of the buffer in which the broadcast message is to be placed.

`type` – `type` is the message type used when sending and receiving the individual messages that constitute the broadcast.

`root` – `root` is the processor ID number of the source of the broadcast message. Note that `root` can be the host.

Output parameters

None

Discussion

Used in conjuction with `recv0`, this routine broadcasts data from one processor, designated as the `root`, to every node processor declared *in use* during the most recent call to `setarc0`. Note that the host or any node processor currently *in use* can be the `root`. To perform a global broadcast from a node processor, each node processor *in use*, except the `root`, must first call `recv0` to receive the broadcast data, and then each node processor *in use*, including the `root`, must call `bcast1` to send the data on to other processors. `bcast1` issues none of the calls to `recv0` that constitute the "receiving half" of a broadcast; it performs only the calls to `send0` that constitute the "sending half" of the operation. When using `recv0` and `bcast1` to perform the broadcast, the user's program need not wait until its participation in the broadcast is over before performing computation with the new data. After using `recv0` to receive the broadcast data, the user's program can use the new data to generate results and send them where they are urgently needed, before resuming its participation in the broadcast with a call to `bcast1`. Typical use of the routine is illustrated below.

```
who0(&numproc, &me, &host);
⋮
if (root != me) recv0(buf, bytes, type);
Use new data in *buf to compute new results.
Send new results where they are urgently needed.
bcast1(buf, bytes, type, root);
```

In contrast, `bcast0` contains both the `recv0`'s and `send0`'s that constitute the broadcast. While its use is more straightforward, `bcast0` gives a processor no opportunity to use the data until after it has completed its participation in the broadcast.

As the processors enter `recv0` and, subsequently, `bcast1`, they cooperate in moving the data from the `root` to the other node processors *in use*, using the interconnection topology set during the most recent call to `setarc0`. After the last processor has exited `bcast1`, every node processor *in use* contains the broadcast data in the buffer pointed to by `buf`.

When a node processor is the `root`, the host does not participate in any way in the broadcast. Indeed, the host can never receive broadcast data; only node processors *in use* can receive broadcast data. Thus, it never makes sense to call `bcast1` on the host with `root` set to a node processor ID number. We have nevertheless found it convenient to allow the host to be the `root` of a broadcast. To broadcast from the host, the host must call only `bcast1`, while every node processor *in use* must first call `recv0`, then `bcast1`. When this is done, the host first sends the broadcast data to node 0, after which the broadcast proceeds just as if it were initially `rooted` at node 0. (Note that the host ID number is a large negative number that can be obtained from `open0` or `who0`. For more details consult the manual pages for these routines.)

A call to `bcast1` must be preceded by a call to `open0` to open interprocessor communication, and by a call to `setarc0` to declare the number of processors *in use* and to indicate what interconnection topology to use. For details on how the various architecture options affect the way the broadcast is performed, consult the manual page for `setarc0`.

Valid values for the input parameters `bytes` and `type` are machine-dependent. Though the routine `bcast1` performs no machine-dependent parameter checking, these two parameters are routinely checked for validity in the low-level communication routine `send0` to which it is passed unchanged. For each target machine, the range of valid values for each of these two parameters is given in section 7.

As with most of the high-level communication functions, certain rules for "consistency" among the input parameters must be followed to ensure successful completion of the operation. First, the user's code must make sure that each non-`root` processor provides a buffer large enough to hold the broadcast message. In particular, `bytes` in `bcast1` on the `root` processor must not exceed `bytes` in `recv0` on any non-`root` processor *in use*. If this condition is not met, then at least one

processor will fail inside `recv0` because there is insufficient buffer space to handle the incoming broadcast message (or on some systems the message is simply truncated if the buffer provided is too small). Second, since the input parameter `type` is used to identify the individual messages that constitute a particular broadcast, each processor must call `recv0` and `bcast1` with the same `type` parameter. Failure to do so is likely to result in one or more processors "hanging" inside `recv0` because an expected incoming broadcast message is missing.

During the execution of the `recv0` that precedes a call to `bcast1`, it is important that the message queue on each processor *in use* not contain a "non-broadcast" message with the same message type as that used by the `bcast1`. If this call to `recv0` happens to receive such a non-broadcast message, then the results are unpredictable. The simplest way to avoid this problem is to avoid "reusing" the broadcast `type` value as a message type anywhere else in the code. It is also important that a broadcast message generated by a call to `bcast1` be removed from the message queue only by the call to `recv0` associated with the call to `bcast1`. Thus it is dangerous when the `recv0/bcast1` pair is immediately preceded by a call to `recv0` with a `type` parameter whose value is -1. In this situation, the "promiscuous" receive is likely to receive the broadcast message, causing the processor to "hang" in the receive portion of the `recv0/bcast1` pair due to the missing message. Not only is the routine unable to recover from this error, it is not even able to identify it and issue an appropriate error message.

gand0, gmax0, gmin0, gor0, gprod0, gsum0, gxor0

These routines compute the component-wise "and", "maximum", "minimum", "or", "product", "sum", and "exclusive or", respectively, of a distributed set of vectors.

routine name	function type	binary operation		valid data types
		name	definition	
gand0	logical	and	$1 \wedge 1 = 1, 1 \wedge 0 = 0,$ $0 \wedge 1 = 0, 0 \wedge 0 = 0$	$0, 1, 2, 3$
gmax0	arithmetic	maximum	$c = \max(a, b)$	$0, 1, 2, 3, 4, 5$
gmin0	arithmetic	minimum	$c = \min(a, b)$	$0, 1, 2, 3, 4, 5$
gor0	logical	or	$1 \vee 1 = 1, 1 \vee 0 = 1,$ $0 \vee 1 = 1, 0 \vee 0 = 0$	$0, 1, 2, 3$
gprod0	arithmetic	multiplication	$c = ab$	$1, 2, 3, 4, 5$
gsum0	arithmetic	addition	$c = a + b$	$1, 2, 3, 4, 5$
gxor0	logical	exclusive or	$1 \oplus 1 = 0, 1 \oplus 0 = 1,$ $0 \oplus 1 = 1, 0 \oplus 0 = 0$	$0, 1, 2, 3$ $0, 1, 2, 3$

Environment

host, node

Synopsis

```
void g__0(buf, items, datatype, msgtype, root)
char *buf;
int items, datatype, msgtype, root;
```

Input parameters

`buf` – **buf** points to the first element of a vector. *Warning: This vector will be overwritten.*

`items` – **items** is the number of elements in the vector.

`datatype` – **datatype** indicates the data type of the elements of the vector:

0 = **char**,

1 = **short**,

2 = **int**,

3 = **long**,

4 = **float**,

5 = **double**.

`msgtype` – **msgtype** is the message type used when sending and receiving the individual messages generated by the global combining operation.

`root` – **root** is the processor ID number of the destination of the final result vector. Note that **root** can be the host.

Output parameters

`buf` – On the **root** processor *only*, **buf** points to the buffer containing the vector resulting from the global combining operation. On the other processors, the contents of the buffer pointed to by **buf** is overwritten with intermediate results.

Discussion

The routines documented here use various binary operations to compute a single vector from a set of vectors distributed among the processors declared *in use* by the most recent call to **setarc0**. Each of the routines assumes the following situation. Let P be the number of node processors declared *in use* during the most recent call to **setarc0**. Let $v^0, v^1, \ldots, v^{P-1}$ be a set of P vectors, each with the same number of elements (**items**) and of the same data type (**datatype**).

Assume that the vectors are distributed among the node processors *in use*, with each processor "owning" a unique vector from the set. Let g__0 represent one of the routines, and let $\odot$ represent the binary operator associated with g__0. Then g__0 computes the vector u whose i-th component is

$$u_i = v_i^0 \odot v_i^1 \odot v_i^2 \odot \cdots \odot v_i^{P-1} ,$$

and stores u on the `root` processor. For example, the routine gsum0 is used to sum the vectors $v^0, v^1, v^2, \ldots, v^{P-1}$ and store the resulting sum on the `root` processor.

The table at the beginning of this section lists all of the "global combining" routines, along with the name and a precise definition of the binary operation on which it is based. This table contains all the information specific to a particular "global combining" routine that the user needs to know. The discussion below presents only material that applies to all of the routines documented here. Throughout we will continue to refer to an arbitrary "global combining" routine by g__0

To "globally combine" a distributed set of vectors and store the result on a node processor, each node processor *in use* must call g__0 with the appropriate input parameters. As the processors enter g__0, they cooperate in combining the data and moving partial results from all node processors *in use* in toward the `root`, using the interconnection topology set during the most recent call to `setarc0`. Upon exit from g__0, the `root` processor contains the final result in the buffer pointed to by `buf`.

When a node processor is the `root`, the host does not participate in the "global combining" operation. Thus, it never makes sense to call g__0 on the host with `root` set to a node processor ID number. We have nevertheless found it convenient to allow the host to be the `root` of a combining operation in the sense that the result is sent there. But the host *never* "owns" a vector to be "combined" with those on the nodes. If the host is the `root`, then g__0 performs exactly as if the `root` were node 0 until the result has been calculated. At this point, the result is sent from node 0 to the host. (Note that the host ID number is a large negative

number that can be obtained from **open0** or **who0**. For more details consult the manual pages for these routines.)

A call to **g__0** must be preceded by a call to **open0** to open interprocessor communication, and by a call to **setarc0** to declare the number of node processors *in use* and to indicate what interconnection topology to use. For details on how the various architecture options affect the way the "combine" operation is performed, consult the manual page for **setarc0**.

Valid values for the input parameters **items** and **msgtype** are machine-dependent. Though the routine **g__0** performs no machine-dependent parameter checking, these two parameters are routinely checked for validity in the low-level communication routines **recv0** and **send0**. The parameter **msgtype** is passed unchanged to these routines; the parameter **items** is scaled to obtain the number of bytes required to store the vector, and that number is checked for validity when it is passed to the **send0** routine. To determine the range of valid values for each of these two parameters, consult Section 7.

As with most of the high-level communication routines, certain rules for "consistency" among the input parameters must be followed to ensure successful completion of the operation. First, the user must make sure that every participating processor calls **g__0** with the same values in **items** and **datatype**. In other words, the vector lengths and data types have to be the same on every processor. If they are not, then it is likely that at least one processor will fail inside the native receive routine because there is insufficient buffer space to hold an incoming vector (or on some systems the message is simply truncated if the buffer provided is too small). Second, since the input parameter **msgtype** is used to identify the individual messages generated by the "combining' operation, each processor must call **g__0** with the same **msgtype** parameter. Failure to do this is likely to result in one or more processors "hanging" inside **g__0** because an expected incoming vector is missing.

During the execution of **g__0**, it is important that the message queue on each processor not contain a message with type **msgtype** from some source other than the combining operation. If **g__0** happens to receive a message not intended for it, then the results are unpredictable. The simplest way to avoid this problem is

to avoid "reusing" a combining operation `msgtype` as a message type anywhere else in the code. It is also important that a "combining" message generated by a call to g__0 be removed from the message queue only by a call to g__0. Thus it is dangerous when a call to g__0 is immediately preceded by a call to `recv0` with a `type` parameter whose value is -1. In this situation, the "promiscuous" receive is likely to receive the "combining" message, causing the processor to "hang" in g__0 due to the missing message. Not only is the routine unable to recover from this error, it is not even able to identify it and issue an appropriate error message.

gcomb0

`gcomb0` *computes a specified component-wise binary combination of a distributed set of vectors.*

Environment

host, node

Synopsis

```
void gcomb0(buf, items, datatype, msgtype, root, comb)
char *buf;
int items, datatype, msgtype, root;
void (*comb)();
```

Input parameters

`buf`	–	`buf` points to the first element of a vector. *Warning: This vector will be overwritten.*
`items`	–	`items` is the number of elements in the vector.
`datatype`	–	`datatype` indicates the data type of the elements of the vector: 0 = `char`, 1 = `short`, 2 = `int`, 3 = `long`, 4 = `float`, 5 = `double`.
`msgtype`	–	`msgtype` is the message type used when sending and receiving the individual messages generated by the global combining operation.
`root`	–	`root` is the processor ID number of the destination of the final result vector. Note that `root` can be the host.

comb – comb is (a pointer to) a binary "combining" routine that operates on the indicated data type. *The operation defined by* comb *must be associative and commutative in order for the result to be well-defined independent of the virtual multiprocessor and of the order of the vectors.*

Output parameters

buf – On the root processor *only*, buf points to the buffer containing the vector resulting from the global combining operation. On the other processors, the contents of the buffer pointed to by buf is overwritten with intermediate results.

Discussion

The routine gcomb0 uses the routine comb to compute a single vector from a set of vectors distributed among the processors declared *in use* by the most recent call to setarc0. gcomb0 assumes the following situation. Let P be the number of node processors declared *in use* during the most recent call to setarc0. Let $v^0, v^1, \ldots, v^{P-1}$ be a set of P vectors, each with the same number of elements (items) and of the same data type (datatype). Assume that the vectors are distributed among the node processors *in use*, with each processor "owning" a unique vector from the set. Let $\odot$ represent the binary operator that is implemented by the function comb. Then gcomb0 computes the vector u whose i-th component is

$$u_i = v_i^0 \odot v_i^1 \odot v_i^2 \odot \cdots \odot v_i^{P-1} ,$$

and stores u on the root processor. For example, if comb sums two numbers, then the call to gcomb0 is equivalent to the routine gsum0, which, in fact, is how gsum0 is implemented.

The routine comb takes four parameters as follows:

```
void comb(p1, p2, items, datatype)
char *p1, *p2;
int items, datatype;
```

The first two parameters, p1 and p2, are pointers to the beginning of the two vectors to be combined. items specifies the number of element in the vector, and datatype specifies the type of each element. comb then "combines" corresponding elements of the two vectors, overwriting the contents of vector p1. As an example, Figure 5 contains the source code for or0, the routine that is passed to gcomb0 inside of gor0. Note that a switch is used to distinguish between the different data types, and that not all cases are represented. The "OR" function for the other data types was deemed less interesting, and those types are ignored. The user determines which data types are understood by comb, and it is the user's responsibility to use only these data types in his application codes. Also note that the function datasize0 is used to identify the number of bytes in data of type datatype. This is an undocumented internal routine of PICL, and can be used by user-written combining routines.

To "globally combine" a distributed set of vectors and store the result on a node processor, each node processor *in use* must call gcomb0 with the appropriate input parameters. As the processors enter gcomb0, they cooperate in combining the data and moving partial results from all node processors *in use* toward the root using the interconnection topology set during the most recent call to setarc0. Upon exit from gcomb0, the root processor contains the final result in the buffer pointed to by buf.

When a node processor is the root, the host does not participate in the "global combining" operation. Thus, it never makes sense to call gcomb0 on the host with root set to a node processor ID number. We have nevertheless found it convenient to allow the host to be the root of a combining operation in the sense that the result is sent there. But the host *never* "owns" a vector to be "combined" with those on the nodes. If the host is the root, then gcomb0 performs exactly as if the root were node 0 until the result has been calculated. At this point, the result is sent from node 0 to the host. (Note that the host ID number is a large negative number that can be obtained from open0 or who0. For more details consult the manual pages for these routines.)

A call to gcomb0 must be preceded by a call to open0 to open interprocessor communication, and by a call to setarc0 to declare the number of node processors

```
void or0(p1, p2, items, datatype)
char *p1, *p2;
int items, datatype;
{
  char *c1, *c2;
  short *s1, *s2;
  int *i1, *i2, i, bytes, size, datasize0();
  long *l1, *l2;

  size = datasize0(datatype);
  bytes = items*size;
  switch (datatype) {
    case 0:
      for (i = 0; i < bytes; i += size) {
        c1 = (char *)(p1 + i);
        c2 = (char *)(p2 + i);
        *c1 |= *c2;
        }
      break;
    case 1:
      for (i = 0; i < bytes; i += size) {
        s1 = (short *)(p1 + i);
        s2 = (short *)(p2 + i);
        *s1 |= *s2;
        }
      break;
    case 2:
      for (i = 0; i < bytes; i += size) {
        i1 = (int *)(p1 + i);
        i2 = (int *)(p2 + i);
        *i1 |= *i2;
        }
      break;
    case 3:
      for (i = 0; i < bytes; i += size) {
        l1 = (long *)(p1 + i);
        l2 = (long *)(p2 + i);
        *l1 |= *l2;
        }
      break;
    }
}
```

Figure 5: Source code for "combining" operator used to implement **gor0**.

in use and to indicate what interconnection topology to use. For details on how the various architecture options affect the way the "combine" operation is performed, consult the manual page for **setarc0**.

Valid values for the input parameters **items** and **msgtype** are machine-dependent. Though the routine **gcomb0** performs no machine-dependent parameter checking, these two parameters are routinely checked for validity in the low-level communication routines **recv0** and **send0**. The parameter **msgtype** is passed unchanged to these routines; the parameter **items** is scaled to obtain the number of bytes required to store the vector, and that number is checked for validity when it is passed to the **send0** routine. To determine the range of valid values for each of these two parameters, consult Section 7.

As with most of the high-level communication routines, certain rules for "consistency" among the input parameters must be followed to ensure successful completion of the operation. First, the user must make sure that every participating processor calls **gcomb0** with the same values in **items** and **datatype**. In other words, the vector lengths and data types have to be the same on every processor. If they are not, then it is likely that at least one processor will fail inside the native receive routine because there is insufficient buffer space to hold an incoming vector (or on some systems the message is simply truncated if the buffer provided is too small). Second, since the input parameter **msgtype** is used to identify the individual messages generated by the "combining' operation, each processor must call **gcomb0** with the same **msgtype** parameter. Failure to do this is likely to result in one or more processors "hanging" inside **gcomb0** because an expected incoming vector is missing.

During the execution of **gcomb0**, it is important that the message queue on each processor not contain a message with type **msgtype** from some source other than the combining operation. If **gcomb0** happens to receive a message not intended for it, then the results are unpredictable. The simplest way to avoid this problem is to avoid "reusing" a combining operation **msgtype** as a message type anywhere else in the code. It is also important that a "combining" message generated by a call to **gcomb0** be removed from the message queue only by a call to **gcomb0**. Thus it is dangerous when a call to **gcomb0** is immediately preceded by a call to **recv0**

with a **type** parameter whose value is -1. In this situation, the "promiscuous" receive is likely to receive the "combining" message, causing the processor to "hang" in **gcomb0** due to the missing message. Not only is the routine unable to recover from this error, it is not even able to identify it and issue an appropriate error message.

getarc0

`getarc0` *returns the number of processors and the interconnection topology currently being used by the high-level global communication routines.*

Environment

host, node

Synopsis

```
void getarc0(nprocs, top, ord, dir)
int *nprocs, *top, *ord, *dir;
```

Input parameters

None

Output parameters

`nprocs` – `nprocs` points to the number of node processors currently *in use.*

`top` – `top` points to a value that indicates the network interconnection topology to be used:
1 = hypercube,
2 = full connectivity,
3 = unidirectional ring,
4 = bidirectional ring.

`ord` – `ord` points to a value that indicates the order of the nodes in a ring. The same ordering is also used by a broadcast when using the "fully-connected" topology option to determine the order of the `send0`'s that comprise the broadcast.
0 = natural ordering, i.e, $0, 1, 2, 3, \ldots,$ `nprocs` $- 1$,
1 = Gray code ordering, i.e, $0, 1, 3, 2, \ldots$.

`dir` – `dir` points to a value that indicates the orientation of a unidirectional ring. It also indicates whether a broadcast using the "fully-connected" topology option proceeds from back to front or front to back through the list of `send0`'s that comprise the broadcast.

-1 = backward,

1 = forward.

Discussion

The routine `getarc0` returns the "virtual" multiprocessor parameters that are set or returned by `setarc0`. This routine is included so that this information can be conveniently obtained by subroutines that do not call `setarc0`. The output parameters will be meaningless if `setarc0` has not been called before calling `getarc0`.

ginv0

`ginv0` computes the inverse binary reflected Gray code function; i.e., given an integer argument i, `ginv0` computes the integer j such that `gray0`(j) = i.

Environment

host, node

Synopsis

```
int ginv0(i)
int i;
```

Input parameters

`i` – integer whose inverse Gray code value is to be computed.

Output parameters

None

Function value

`ginv0` returns the value of the integer for which the Gray code function is equal to the input integer.

Discussion

This function is useful when embedding rings and meshes in certain architectures, such as hypercubes. In particular, the consecutive members of a Gray code sequence are immediate neighbors in a binary hypercube.

gray0

`gray0` *computes the binary reflected Gray code value of a given integer argument.*

Environment

host, node

Synopsis

```
int gray0(i)
int i;
```

Input parameters

`i` – is the integer argument whose Gray code value is to be computed.

Output parameters

None

Function value

`gray0` returns the Gray code value of the given integer input.

Discussion

This function is useful when embedding rings and meshes in certain architectures, such as hypercubes. In particular, the consecutive members of a Gray code sequence are immediate neighbors in a binary hypercube.

setarc0

`setarc0` *sets the number of processors and the interconnection topology to be used by the high-level global communication routines.*

Environment

host, node

Synopsis

```
void setarc0(nprocs, top, ord, dir)
int *nprocs, *top, *ord, *dir;
```

Parameters

On the *host* each of the four parameters described below is an *input* parameter; On the *node* processors, each is an *output* parameter. The host sends the information to the nodes.

`nprocs` – `nprocs` points to the number of node processors *in use* during subsequent calls to any of the high-level communication routines. This number must be less than or equal to the number of node processors in the user's current allocation (as determined by `open0` or `who0`). A node processor will not return from a call to `setarc0` until the number of processors *in use* is greater than its processor ID or is zero.

`top` – `top` points to a value that indicates the network interconnection topology to be used:
1 = hypercube,
2 = full connectivity,
3 = unidirectional ring,
4 = bidirectional ring.

`ord` – `ord` points to a value that indicates the order of the nodes in a ring. The same ordering is also used by a broadcast when using the "fully-connected" topology option to determine the order of the `send0`'s that comprise the broadcast. `ord` is ignored when the hypercube topology if used (`*top` = 1).
0 = natural ordering, i.e, $0, 1, 2, 3, \ldots, \mathtt{nprocs} - 1$,
1 = Gray code ordering, i.e, $0, 1, 3, 2, \ldots$.

`dir` – `dir` points to a value that indicates the orientation of a unidirectional ring. It also determines whether a broadcast using the "fully-connected" topology option proceeds from back to front or front to back through the list of `send0`'s that comprise the broadcast. `dir` is ignored when `top` = 1.
−1 = backward,
1 = forward.

Discussion

The routine `setarc0` prescribes to the high-level communication routines the "virtual" multiprocessor architecture on which they are to be carried out and, implicitly, the algorithm used to perform the operation. By "virtual" multiprocessor system we mean that there is no assumption that the architecture declared with `setarc0` is *physically realized* by the machine on which the program is running. With automatic routing between arbitrary pairs of processors, algorithms designed for any network topology will work, independent of the network used by the machine. For example, a global broadcast can be programmed as if the machine were fully-connected, though this topology is not physically realized by any of our current target multiprocessing environments. Performing a global broadcast with `bcast0` *after* specifying a fully connected topology (`*top` = 2) with `setarc0` provides the user with such a global broadcast.

To prescribe a "virtual" multiprocessor architecture, *the host and every node processor currently allocated to the user must call* `setarc0`. On the host, the four parameters of `setarc0` are input parameters. These suffice to determine the number of node processors deemed to be *in use*, and the interconnection

topology to be associated with these processors. While in **setarc0**, the host program sends the input parameters to the designated node processors in the user's allocation. Each node processor in this "virtual" multiprocessor records and returns to the user the parameters it receives from the host while executing **setarc0**. If a node processor is not in this "virtual" multiprocessor, then it does not exit from **setarc0**. To release these idled processors before the end of the host program, **setarc0** should be called on the host with ***nprocs** set equal to 0.

The following high-level communication routines should not be used until after **setarc0** has been called: **barrier0**, **bcast0**, **bcast1**, **gand0**, **gcomb0**, **gmax0**, **gmin0**, **gor0**, **gprod0**, **gsum0**, **gxor0**. Only the two routines that perform no interprocessor communication, **ginv0** and **gray0**, do not depend on prior execution of **setarc0**. Also note that before **setarc0** can be called, the user must open interprocessor communication with a call to **open0**. (For more information on how to use **open0** consult its manual page in Section 3.)

The message type used internally by **setarc0** to send the "virtual" multiprocessor parameters to the node processors is machine-dependent. The value used for a given target machines is described in Section 7. A call to **setarc0** on a node may fail if it is immediately preceded by a call to **recv0** with a **type** parameter of -1. In this situation, the "promiscuous" receive may receive the message from **setarc0** on the host, causing the node processor to "hang" in **setarc0** waiting for the now missing message. Not only is the routine unable to recover from this error, it is not even able to identify it and issue an appropriate error message. Thus, it is important that the user's program never allow "promiscuous" receives to interfere with **setarc0** in this way.

During the execution of **setarc0**, it is also important that the message queue of each node processor not contain a "non-setarc0" message with the same message type as that used by **setarc0**. In this case the node processor will not hang, but the message received by **setarc0** may not contain the desired "virtual" multiprocessor parameters, and anomalous behaviour can result. The user can ensure that this does not occur by not using this particular message type. (See Section 7 for the value on a given target machine.) Since the **setarc0** message type is near the high end of the range of legal message types, portability can be enhanced by

keeping user-defined message types smaller than this value for all machines that the program is likely to be executed on.

The remainder of this section is devoted to discussing the input parameters in more detail.

nprocs. When `setarc0` is called on all processors to "set" the architecture, all node processor in the current allocation with a processor ID number higher than `*nprocs` − 1 are disabled until a subsequent call to `setarc0` changes the value of `*nprocs`. After a call to `setarc0`, we refer to the node processors $0, 1, 2, \ldots,$ `*nprocs` − 1 as being *in use*. The high-level communication routines use the value pointed to by `nprocs` to determine which node processors to use to complete the indicated operation.

Historically, the purpose for allowing the number of processors *in use* to vary in this way while the node allocation remains unchanged is to provide a convenient tool for making multiple runs of parallel algorithms on different numbers of processors during a single run of the host and node programs. Typically, unused processors remain "hung" in their call to `setarc0`, awaiting the next case in which they will be used. Since the correct execution of the host and node programs depends on each node program calling `close0` at some point before terminating its execution, the host program should "wake-up" any idled processors before itself calling `close0`. When the host calls `setarc0` with `*nprocs` set to 0, then all node processors calling `setarc0` will return from the call with `*nprocs` set to 0. This special value can be used by an application program to recognize that the use of a "virtual" multiprocessor is no longer in force, and that the high-level communication routines should no longer be used.

Restrictions. Note that if the Gray code ordering is in effect (i.e., `*ord` = 1), then the number of processors *in use* must be an integer power of two. Note also that, contrary to what one might expect, choosing the hypercube option does *not* restrict `*nprocs` to be an integer power of two. The algorithms for the high-level communication routines have been designed to work on "incomplete" hypercubes with any positive number of processors.

Caveats. While the high-level communication routines use the value pointed to by **nprocs** to decide what processors to use, the low-level routines do not. Thus a processor can still use **send0** to send (or **recv0** to receive) a message to (from) a processor whose ID is larger than or equal to ***nprocs**. In most circumstances, this represents a logical error, and the program will not perform correctly. Since a node processor whose ID is larger than or equal to ***nprocs** will "hang" only if it calls **setarc0**, it is possible to specify a virtual multiprocessor for use with the high-level communication routines and still use the other processors. But then it is the user's responsibility to ensure that only the node processors *in use* issue calls to the high-level routines and that any allocated node processors *not in use* issue no calls to any high-level communications routine.

top. We assume the reader is familiar with the network options we have included. Figure 4 shows how the data flows out through the network away from the **root** during a global broadcast. Figure 4 shows how the data flows in through the network toward the **root** during a "global combining" operation, such as computing a global sum, product, minimum, or maximum. In both figures, we assume that processor 0 is the root; in practice, any processor can act as a root. It should be clear from the figures that in our routines data flows along the usual paths of choice through the networks offered by the package. Note that we anticipate offering grids in some future upgrade of PICL.

dir and ord. Together, **dir** and **ord** prescribe how to arrange the node processors into a ring, and how to order the **send0**'s of a global broadcast using the "fully-connected" topology option. In the discussion that follows, P denotes the number of processors *in use*. Note that the **dir** and **ord** parameters are ignored when the hypercube topology option is chosen.

***dir**: forward, ***ord**: natural. In this case a unidirectional ring is given by:

$$0 \rightarrow 1 \rightarrow 2 \rightarrow \cdots \rightarrow (P-1) \rightarrow 0 \,.$$

***dir**: backward, ***ord**: natural. In this case a unidirectional ring is given by:

$$(P-1) \rightarrow (P-2) \rightarrow (P-3) \rightarrow \cdots \rightarrow 2 \rightarrow 1 \rightarrow 0 \rightarrow (P-1) \,.$$

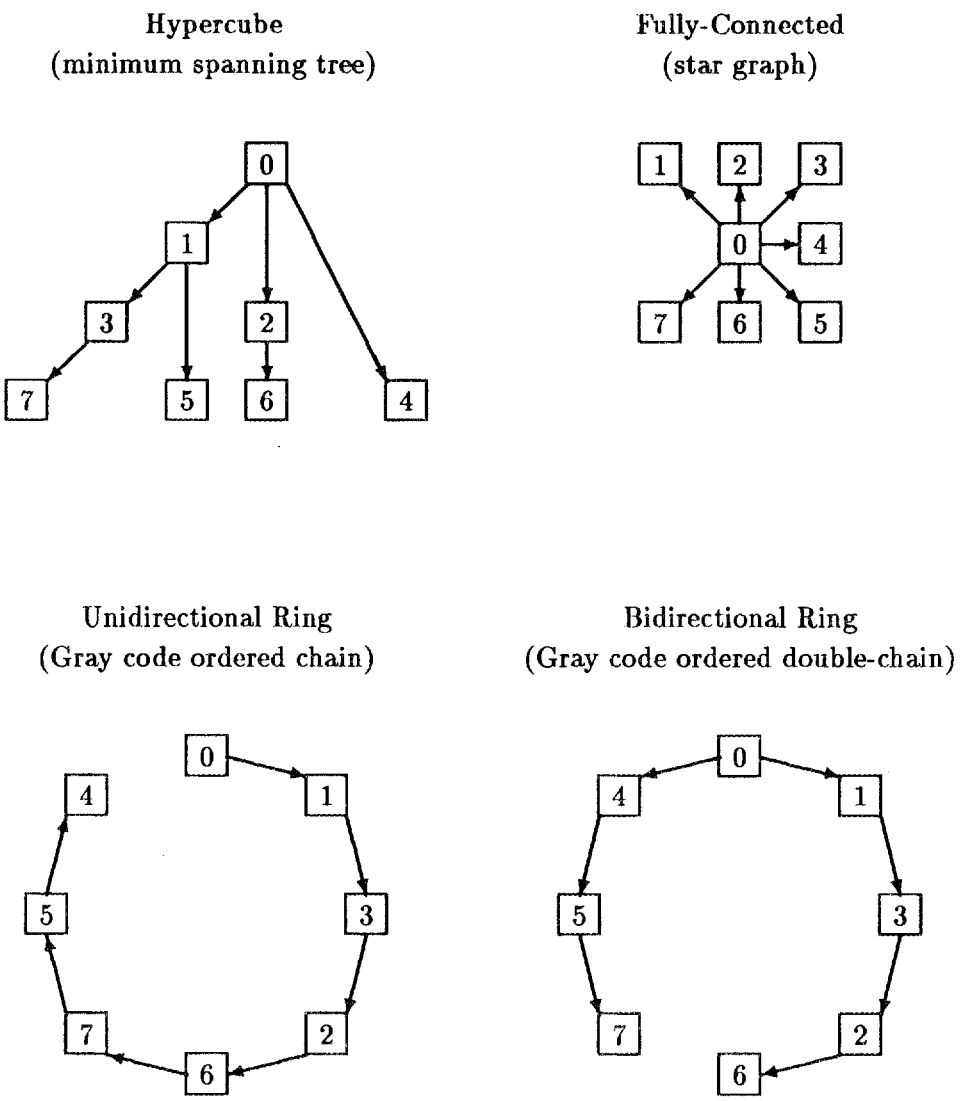

Figure 6: Example data flow of a global broadcast on different interconnection topologies.

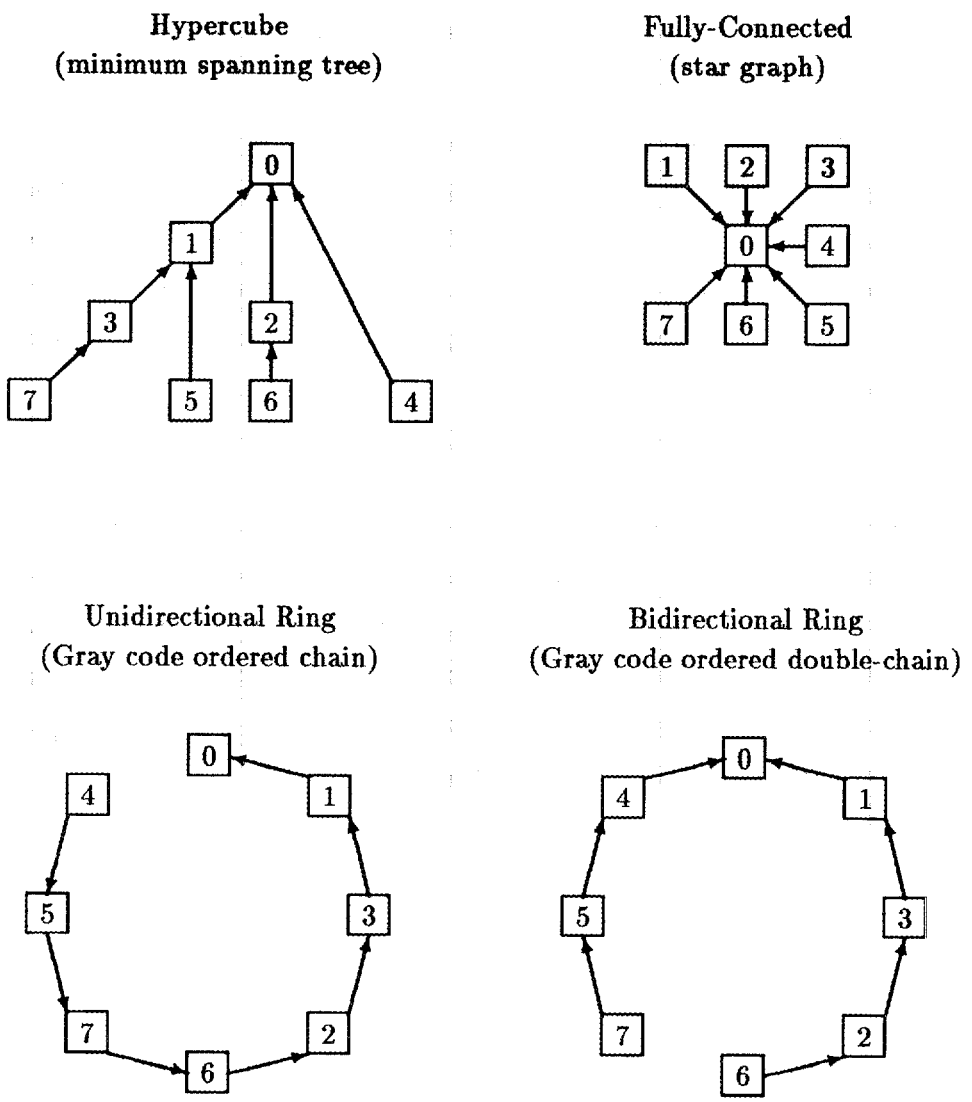

Figure 7: Example data flow of a global "combine" operation on different interconnection topologies.

`*dir`: forward, `*ord`: Gray. In this case the nodes of a unidirectional ring are ordered by their numbering in the Gray code mapping of the processor ID numbers *in use*. When the Gray code option is used, the number of processors *in use*, `*nprocs`, must be an integer power of two. The following shows how this choice for these parameters arranges a unidirectional ring with 8 processors.

$$0 \rightarrow 1 \rightarrow 3 \rightarrow 2 \rightarrow 6 \rightarrow 7 \rightarrow 5 \rightarrow 4 \rightarrow 0 .$$

`*dir`: backward, `*ord`: Gray. In this case the nodes of a unidirectional ring are ordered *in reverse order* by their numbering in the Gray code mapping of the processor ID numbers *in use*. When the Gray code option is used, the number of processors *in use*, `*nprocs`, must be an integer power of two. The following shows how this choice for these parameters arranges a unidirectional ring with 8 processors.

$$4 \rightarrow 5 \rightarrow 7 \rightarrow 6 \rightarrow 2 \rightarrow 3 \rightarrow 1 \rightarrow 0 \rightarrow 4 .$$

In all cases, a bidirectional ring is arranged in the same way as a unidirectional ring, but with bidirectional arrows. The `send0`'s from the `root` of a global broadcast using the "fully-connected" topology option are performed in the same order as that indicated by the associated unidirectional example. For example, if `*dir` = forward, if `*ord` = natural, and if the `root` is processor i, then processor i first sends to processor $i + 1$, then to processor $i + 2$, etc.

5. Execution tracing routines

Purpose

When using the low-level primitive or high-level communication routines, the user can produce time-stamped records that detail the course of execution on each processor. The tracing routines allow the user to control the generation of these trace records. Since *execution tracing can be intrusive*, it is important that the user understand how the tracing routines may alter the behavior of his program.

Machines and Multiprocessing Environments

Implementations of the execution tracing routines are available for the following machines and multiprocessing environments:

iPSC/2 hypercube iPSC/860 hypercube Ncube/3200 hypercube

Synopsis

The host and node programs should be linked, respectively, with host and node libraries containing the tracing routines listed below. All routines except three are available on both host and node. The routines **traceenable** and **tracehost** are available only on the host; the routine **tracenode** is available only on node processors.

```
void traceenable(char *tracefile, int verbose)
void tracehost(int tracesize, int flush)
void tracenode(int tracesize, int flush, int sync)
void traceexit()


void tracelevel(int event, int compstats, int commstats)
void traceinfo(int remaining, int event, int compstats,
               int commstats)
```

```
void tracemark(int marktype)
void tracemsg(char *message)
void traceflush()
```

With one exception, programs that use PICL will *not* fail due to incorrect usage of these routines. While incorrect usage may cause loss of trace information or degradation in performance, it should not cause wrong answers or abnormal termination. The one exception to this rule is that misuse of the synchronization option in `tracenode` will cause deadlock. See the manual pages for `tracenode` for further information.

Standard Usage

To collect trace data for the entire run of an application code, it is sufficient to follow these steps:

1) Insert calls to `traceenable`, `tracelevel`, and `tracehost` at the beginning of the host program. For example,

```
traceenable(''tracefile'', 1);
tracelevel(3, 3, 3);
tracehost(100000, 0);
```

2) Insert calls to `traceexit` and `traceflush` at the end of the host program. For example,

```
traceexit();
traceflush();
```

3) Insert calls to `tracelevel` and `tracenode` at the beginning of the node program. For example,

```
tracelevel(3, 3, 3);
tracenode(100000, 0, 0);
```

If trace data is needed only from the node program, then the steps become even simpler:

1) Insert a call to **traceenable** at the beginning of the host program. For example,

```
traceenable(''tracefile'', 1);
```

2) Insert calls to **tracelevel** and **tracenode** at the beginning of the node program. For example,

```
tracelevel(3, 3, 3);
tracenode(100000, 0, 0);
```

traceenable

`traceenable` *is used to open a trace file and to select one of two available formats for the trace records.*

Environment

host

Synopsis

```
void traceenable(tracefile, verbose)
char *tracefile;
int verbose;
```

Input parameters

`tracefile`	–	`tracefile` points to a character string containing the name of the new trace file to be opened.
`verbose`	–	If `verbose` is 1, then trace records are written into the trace file in "verbose" form, i.e., with a keyword labeling each numeric value in the record. Otherwise, each trace record is simply a sequence of numerical values. information.

Output parameters

None

Discussion

The `traceenable` routine is used to open a new trace file and to specify whether trace records should be written in the *verbose* or *compact* format. Trace records written in the verbose format are easy to read because each data item is preceded by a descriptive keyword. However, verbose trace files require approximately three times more disk space than compact files. Though more difficult to read because they contain no keywords, compact trace files can always be transformed

into verbose trace files later. See Section 6 for instructions on how to do this. The compact option can be very helpful if one must produce trace files when disk space is at a premium.

`traceenable` also prepares the host to receive and process trace information as it arrives from the nodes. Without a prior call to `tracenable`, the host simply disposes of incoming trace records, and such records disappear without being written into a trace file. To avoid losing trace data, it is good practice to call `traceenable` before executing `load0`. In general, it is good practice to make `traceenable` one of the first executable instructions in the host program any time execution tracing is to be used on the host or on any node.

Until `traceenable` is called, calling the following tracing routines on the host does nothing: `tracehost`, `traceexit`, `traceflush`, `tracemark` and `tracemsg`. The only exceptions are `tracelevel` and `traceinfo`, whose function do not depend in any way on prior execution of `traceenable`.

Multiple calls to `traceenable` in a host program can be used to change the file to which the trace information is written. This is useful when a single run of the host program generates either multiple runs on the nodes or a distinct sequence of computational phases on the nodes that need to be analyzed separately. In either situation it is often convenient, if not necessary, for the trace information from separate runs or computational phases to be stored in separate trace files. Some care is required to make sure that the host program switches from one trace file to another only after the first file is complete. Use the following technique to ensure a clean switch. Immediately after the end of the current run or phase *on the nodes*, use `sync0` to synchronize *all* the nodes. Next, issue a call to `traceexit`, followed by a call to `traceflush`. Finally, have each node send a message to the host, indicating that it has finished sending its trace information to the host, and call `tracenode` again to restart the collection of trace data. The host program should wait until it has received all of the messages from the nodes before executing `traceenable` again to set up a new trace file.

traceexit

`traceexit` *stops the collection of trace information.*

Environment

host, node

Synopsis

```
void traceexit()
```

Input parameters

None

Output parameters

None

Discussion

On the host, the routines `traceenable` and `tracehost` are used to start the collection of trace information, while on the nodes, `tracenode` alone performs this function. Once these routines have been executed, code is activated in the low-level and high-level PICL routines to accumulate busy/idle time and communication statistics, such as number and total volume of the messages sent and received. They also enable the user to activate (with a call to `tracelevel`) code that records specified events in the trace file. Calling `traceexit` deactivates the accumulation of statistics and the ability to generate trace records.

If tracing has been initiated, `traceexit` places a "traceexit" record in the trace buffer. On the nodes, `traceexit` is automatically called within `close0` if it is not called explicitly in the node program, and a "traceexit" record will always be generated. On the host, `close0` *does not* call `traceexit`, and a "traceexit" record is not automatically generated.

After calling `traceexit`, calling the following tracing routines does nothing: `tracemark`, `tracemsg`, `traceexit`, `tracehost` (on the host) and `tracenode` (on

a node). The routine `traceflush` can be called one more time before it, too, is disabled. This allows the user to place a `traceexit` record at the end of the trace data before using `traceflush` to send trace information to the trace file.

One of the primary features associated with the `traceexit` routine is that it can be used on the nodes in conjuction with the `traceenable` and `traceflush` routines to create more than one trace file during a single run. How this can be done is described in the manual pages for `traceenable`.

traceflush

`traceflush` *writes the local trace information to the trace file.*

Environment

host, node

Synopsis

```
void traceflush()
```

Input parameters

None

Output parameters

None

Discussion

The routine `traceflush` writes into the trace file all trace records currently held in the user-supplied trace buffer. Under certain circumstances discussed below, it is an important tool for controlling when node trace data will be sent to the host to be printed into the trace file. This control should be handled judiciously, since an interprocessor communication network saturated with a high volume of trace data can seriously degrade performance when useful computation is in progress *at the same time.*

It is important to realize that *on the nodes* it is usually not necessary or desirable to use `traceflush` to write the trace data into the trace file. The simplest and most straightforward way to use the trace facility is to generate a single trace file that records the behavior of a single run on the host and on the nodes. When the trace facility is used in this manner, no explicit calls to `traceflush` are necessary *on the nodes* to write the data to the file. The routine `close0` automatically

performs this function before it closes interprocessor communication. The host and node versions of `close0` cooperate to handle this task as follows. After entering `close0`, the node program waits until `close0` is executed on the host, at which time the host will prompt the node for the trace information. Then, and only then, does it send the trace data to the host. Moreover, since `close0` on the host does not prompt for trace information until *all* nodes have entered `close0`, the nodes will not contaminate the network with trace data while useful computation is in progress.

However, when using the execution tracing facility to create more than one trace file in a single run, the user must explicitly control the movement of trace data from the nodes to the trace file. The manual page for `traceenable` describes how this is done and what role `traceflush` plays in the procedure.

On the nodes, there are other cases when the trace information must be sent back before `close0` is executed. For example, a programming error in a node program may cause it to terminate before it reaches `close0`. To gain access to trace data for debugging purposes, the user must call `traceflush` on the node before failure occurs. Another reason for calling `traceflush` is a full (or nearly full) trace buffer. The user's program can call `traceinfo` to find out how much space remains available in the trace buffer. If there is insufficient space to hold the trace records generated by the next computational phase, then it may be best to flush the contents of the trace buffer to the trace file between phases, rather than allowing the trace data to flood the network in the middle of the next phase (assuming automatic flushing is enabled) or losing trace information because the buffer fills up.

On the host, execution of `close0` does not automatically route the host's trace data to the trace file as it does on a node processor. The routine `traceflush` *must* be called to do this; otherwise, the trace data is simply lost.

Since `traceflush` can significantly degrade the performance of both node and host programs, the user should carefully choose when `traceflush` is to be executed. In general, if performance measurements are being made it is best to synchronize all processors with a call to `sync0` before a call to `traceflush` in order to make sure that the execution of `traceflush` on one processor does not

flood the network with trace data that will degrade the performance of other processors still doing useful work.

tracehost

`tracehost` *starts the collection of trace information on the host.*

Environment

host

Synopsis

```
void tracehost(tracesize, flush)
int tracesize, flush;
```

Input parameters

`tracesize` – `tracesize` is the size (in bytes) of the buffer to be allocated for collecting trace records in main memory. If it is too small to record the minimum amount of trace information, a default amount will be allocated.

`flush` – `flush` indicates whether or not the trace buffer should be *automatically flushed* when the buffer fills up. If `flush` is 1, then all data in the trace buffer is immediately written to the trace file when the buffer fills up. If `flush` contains any value other than 1, then the generation of trace records stops after the buffer fills up.

Output parameters

None

Discussion

The routine `tracehost` is used to start the collection of trace information on the host. Once `tracehost` has been executed, code is activated in the *host version* of the low-level and high-level PICL routines to accumulate busy/idle time statistics and communication statistics, such as the number and volume of messages sent

and received. Calling `tracehost` also enables the user to activate (with a call to `tracelevel`) code that records specified events in the trace file. `tracenode` performs virtually the same functions on the nodes.

Host trace records are collected in the main memory of the host before they are written in the trace file. The parameter `tracesize` is used to determine the size of the trace buffer `malloc`'ed by `tracehost` for this purpose. It is usually best if `tracesize` is large enough to allow the trace buffer to contain all trace records the host processor generates during the run. Of course, it must also be small enough to allow the executable to be loaded on the host. Trace records (stored in memory) average between five and six long integers in length. Thus, if the user has a rough idea how many trace records his code will generate during a given run, it is easy to calculate a reasonable upper bound on the required size of the buffer. Note that a minimum amount of buffer space is required for tracing to work, and at least this amount will be allocated no matter how much is specified by `tracesize`. Currently, this minimum amount is 208 bytes.

The `flush` parameter prescribes what action to take if the trace buffer becomes full before the host program ends. When `flush` is 1, the contents of the full trace buffer are immediately written into the trace file and tracing continues as before; otherwise, the generation of trace records is halted until the contents of the buffer are explicitly written into the trace file via a call to the `traceflush` routine. Note that the *calculation* of the performance statistics continues in either case. As a rule, `flush` should be set to 1 only when debugging. It should definitely *not* be set to 1 when performance is being measured.

On the host, both `traceenable` and `tracehost` must be called before tracing will begin. Calling `tracehost` does nothing unless there has been a prior call to `traceenable`, and the following routines do nothing unless both `traceenable` and `tracehost` have been called (in that order): `traceflush`, `tracemark`, `tracemsg`, and `traceexit`. The only exceptions are `tracelevel` and `traceinfo`, whose function do not depend in any way on prior execution of either `traceenable` or `tracehost`.

Note that tracing on the host is completely independent of tracing on the nodes. For example, the user may

1) start tracing on the host and on one or more node processors,

2) start tracing on the host but leave tracing "off" on all node processors,

3) leave tracing "off" on the host but start tracing on one or more node processors, or

4) leave tracing "off" on the host and on all node processors.

traceinfo

`traceinfo` *returns the current values assigned to the parameters that control the type and amount of trace data generated.*

Environment

host, node

Synopsis

```
void traceinfo(remaining, event, compstats, commstats)
int *remaining, *event, *compstats, *commstats;
```

Input parameters

None

Output parameters

`remaining`	–	`remaining` points to a a lower bound on the number of trace records that can be added to the trace buffer before it becomes full.
`event`	–	`event` points to a value that indicates which events will generate "event" trace records.
`compstats`	–	`compstats` points to a value that indicates which events will generate cumulative busy/idle time records.
`commstats`	–	`commstats` points to a value that indicates which events will generate communication statistics records.

Discussion

The `traceinfo` routine returns the current values of the variables used to determine which events generate trace records. See the manual pages for `tracelevel` for a detailed description of what these values mean. There are four distinct types of trace records generated, and three of these have their own "level" indicators to

control the level of detail recorded. The meaning of the level values are essentially the same for all three types, except that an "event" trace record is always generated for calls to `tracehost`, `tracenode`, `tracelevel`, `traceflush`, `traceexit`, `open0`, and `close0`. The routine `traceinfo` also returns a lower bound on how many more trace events can be recorded before the trace buffer becomes full. A full description of the types of trace records is given in the discussion of the trace data format in Section 6.

tracelevel

`tracelevel` *sets the parameters that control the type and amount of trace data generated.*

Environment

host, node

Synopsis

```
void tracelevel(event, compstats, commstats)
int event, compstats, commstats;
```

Input parameters

`event` – `event` determines which events will generate "event" trace records.

`compstats` – `compstats` determines which events will generate cumulative busy/idle time records.

`commstats` – `commstats` determines which events will generate communication statistics records.

The following table interprets the values that the `event` "level" variable can take on.

`level value`	events to be traced
all values	includes calls to `tracehost`, `tracenode`, `tracelevel`, `traceflush`, `traceexit`, `open0`, and `close0`
≥ 1	also includes calls to `tracemark`
≥ 2	also includes calls to any high-level communication routine and calls to `send0` and `recv0` *not* issued by a high-level communication routine.
≥ 3	also includes calls to `send0` and `recv0` from within a high-level communication routine.

The following table interprets the values that the other two trace "level" variables, **compstats** and **commstats**, can take on.

level value	events to be traced
≤ 0	none
≥ 1	includes calls to **tracemark**, **tracelevel**, open0, and close0
≥ 2	also includes calls to any high-level communication routine and calls to **send0** and **recv0** *not* issued by a high-level communication routine
≥ 3	also includes calls to **send0** and **recv0** from within a high-level communication routine

Output parameters

None

Discussion

The **tracelevel** routine sets the variables that determine which events generate trace records. There are four distinct types of trace records generated, and three of these have their own "level" indicators to control the level of detail recorded. The meaning of the level values are essentially the same for all three types, except that an "event" trace record is always generated for calls to **tracehost**, **tracenode**, **tracelevel**, **traceflush**, **traceexit**, open0, and close0. A full description of the types of trace records is given in the discussion of the trace file format at the beginning of this section.

Note that the number of trace records generated by a single PICL subroutine call varies from routine to routine. For example, a single call to **recv0** often generates two "event" trace records: a **recv_blocking** record and a **recv_waking** record, but a call to open0 always generates a single "event" trace record. Thus, there is not a one-to-one correspondence between trace records and PICL subroutine calls.

Tracing is started on the host by calling both **traceenable** and **tracehost**; it is started on a node processor by calling **tracenode**. On host or node, if tracing

has not been started or if it has been disabled with a call to `traceexit`, then `tracelevel` still records the input parameters for possible future use. Thus, the routine can be used to set the "level" parameters at any time during a program's execution.

tracemark

tracemark *is used to "mark" a user-specified event in the trace buffer.*

Environment

host, node

Synopsis

```
void tracemark(type)
int type;
```

Input parameters

type – **type** is an identifying integer printed in the trace record generated by this routine. It should be a user-chosen value that signifies the occurence of some user-specified event.

Output parameters

None

Discussion

The **tracemark** routine is used to record a user-defined event. The actual trace records generated by **tracemark** depend on the tracing levels requested by the most recent call to **tracelevel**. A call to **tracemark** can generate a record showing the total busy/idle time at the time of the call, a record showing the cumulative send/receive message counts and volumes at the time of the call, and/or a time-stamped record of the call to **tracemark** containing the associated **type**, which should be used to identify the event signified by the record. If more than one integer field is needed to specify the event, then two or more successive calls to **tracemark** can be used to fully specify what happened.

On the host, calling this routine does nothing unless tracing has been started with successive calls to **traceenable** and **tracehost**. On a node, calling **tracemark**

does nothing unless tracing has been started with a call to `tracenode`. On both host and node, a call to `traceexit` disables `tracemark`.

tracemsg

tracemsg *writes a string into the trace file immediately.*

Environment

host, node

Synopsis

```
void tracemsg(string)
char *string;
```

Input parameters

string – **string** points to a character string to be printed in the trace file. Only the first 80 characters will be printed.

Output parameters

None

Discussion

Used primarily for debugging, **tracemsg** sends a string to the trace file immediately. When used in conjunction with **sprintf** (used to fill a character buffer) it provides the same functionality as **fprintf**, but for *small messages only* ($\leq$ 80 characters). In particular, a node processor does *not* wait until the local trace buffer is flushed before sending the message to the host to be written in the trace file. However, for purposes other than debugging, **tracemsg** should be used sparingly to avoid excessive network traffic that can degrade performance. Our experience indicates that moderate use of **tracemsg** does not noticeably degrade performance.

tracenode

`tracenode` *starts the collection of trace information on a node processor.*

Environment

node

Synopsis

```
void tracenode(tracesize, flush, sync)
int tracesize, flush, sync;
```

Input parameters

`tracesize` – `tracesize` is the size (in bytes) of the buffer to be allocated for the collection of trace records in main memory. If it is too small to record the minimum amount of trace information, a default amount will be allocated.

`flush` – `flush` indicates whether the trace buffer should be *automatically flushed* when the buffer is full. If `flush` is 1, then all data in the trace buffer is immediately written to the trace file when the buffer fills up. If `flush` contains any value other than 1, then the generation of trace records stops after the buffer fills up.

`sync` – If `sync` is 1, then the node processors synchronize before continuing. This permits the trace record time-stamps on the different processors to be normalized with respect to the same approximate starting time. All node processors must use the same value for `sync`; otherwise, some of the processors may never return from the call to `tracenode`.

Output parameters

None

Discussion

The routine `tracenode` is used to start the collection of trace information on a node. Once `tracenode` has been executed, code is activated in the *node version* of the low-level and high-level PICL routines to accumulate busy/idle time and communication statistics, such as number and volume of messages sent and received. Calling `tracenode` also enables the user to activate (with a call to `tracelevel`) code that records specified events in the trace file. `tracehost` performs virtually the same functions on the host.

Node trace records are collected in the main memory of the node processor before they are sent to the host to be written in the trace file. The parameter `tracesize` is used to determine the size of the trace buffer `malloc`'ed by `tracenode` for this purpose. It is usually best if `tracesize` is large enough to allow the trace buffer to contain all trace records the node processor generates during the run. Of course, it must also be small enough to allow the executable to be loaded and run on a node processor. Trace records (stored in memory) average between five and six long integers in length. Thus, if the user has a rough idea how many trace records his code will generate during a given run, it is easy to calculate a reasonable upper bound on the required size of the buffer. Note that a minimum amount of buffer space is required for tracing to work, and at least this amount will be allocated no matter how much is specified by `tracesize`. Currently, this minimum amount is 208 bytes.

The `flush` parameter prescribes what action to take if the trace buffer becomes full before the node program ends. When `flush` is 1, the contents of the full trace buffer are immediately sent to the host and tracing continues as before; otherwise, the generation of trace records is halted until the contents of the buffer are explicitly sent to the host via a call to the `traceflush` routine. Note that the *calculation* of the performance statistics continues in either case. As a rule, `flush` should be set to 1 only when debugging. It should definitely *not* be set to 1 when performance is being measured.

When `sync` is 1, the node processors are synchronized before they exit from `tracenode`. As with `sync0`, *all node processors must participate in the synchro-*

nization. If some processors request synchronization while others do not, then the "synchronizing" processors will never exit the call to `tracenode`.

It is often desirable to compare the time-stamps of two trace records generated by different processors. The fact that each processor has its own independent clock makes it difficult to produce time-stamps on different processors whose comparison is guaranteed to be very meaningful. The primary use of the `sync` option is to attempt to "sychronize" time-stamps from different node processors. Whether or not the `sync` option has been used, the time-stamps produced on each node processor are normalized by subtracting off the time at which `tracenode` is called. When the `sync` option is used, the processors are approximately synchronized immediately before each determines the time it will use to normalize subsequent time-stamps, and thus, all subsequent time-stamps are normalized against the same *approximate* starting time. Comparability of time-stamps then depends both on the amount of "clock drift" during the computation and on the sharpness of the synchronization technique (which is machine-dependent). Note that PICL does not provide an option for automatically synchronizing the time-stamps produced on the host with those produced on the nodes.

Tracing may be enabled on *any subset of the allocated nodes.* In other words, an arbitrary subset of the allocated nodes can call `tracenode`, while the remaining nodes do not, provided, of course, none of the calls to `tracenode` invoke the `sync` option. Only nodes that call `tracenode` will generate trace data. For example, tracing only a single node sometimes provides sufficient information to understand the performance of a parallel algorithm, while generating much less trace data (which can be important).

Note also that tracing on the host is completely independent of tracing on the nodes. For example, the user may

1) start tracing on the host and on one or more node processors,
2) start tracing on the host but leave tracing "off" on all node processors,
3) leave tracing "off" on the host but start tracing on one or more node processors, or
4) leave tracing "off" on the host and on all node processors.

6. Trace data format

There are two possible formats for the trace file: *verbose* and *compact*. Both are ASCII, but the verbose format can be more easily read because the type of each trace record is identified by a keyword, and all numeric data is preceded by a descriptive keyword. The compact format has no keywords. The record type in the compact format is represented by an integer, and the data items are in the same order as in the verbose format, but without the keywords. While the compact format is more difficult to read, it takes up approximately one third the space required by the verbose format, and it is simple to transform a trace file from a one format to the other. The choice of trace file format is made via the `verbose` parameter in the `traceenable` command. See the manual pages for `traceenable` for a complete discussion of this parameter.

There are four types of trace records: *event*, *computation statistics*, *communication statistics*, and *trace messages*. All four types have a time-stamp indicating when they were generated. All time-stamps and other measures of time are displayed as two integer values, the first representing the number of seconds and the second representing the number of microseconds that have elapsed since tracing was initiated. They also have a field identifying the processor on which they were generated. If the processor is a node, then the identifying integer is between 0 and $P-1$, where P is the number of node processors allocated. If the processor is the host, then the identifying integer is $-32,768$.

6.1. Event records

The event records are either associated with a specific PICL routine or mark an unavoidable change of behavior in the tracing (like the trace buffer filling up). Whether or not a given routine generates the associated trace record is a function of the value of the `event` parameter in `tracelevel`. See the manual pages for `tracelevel` for details on how different values of this parameter prescribe different levels of event tracing. The following list describes the verbose format event trace records:

- The `trace_start` record marks when tracing begins on the host or on a node processor. It is generated by the `tracehost` and `tracenode` routines. It has a clock field that is always zero since all time-stamps are normalized with respect

to the time at which tracing was initiated. It also has fields for the tracing level parameters that are in effect at the time that tracing starts. See the manual pages for `tracelevel` for a description of the meaning of these values. Example:

```
trace_start  clock 0 0 node 17 event 0 compstats 0 commstats 0
```

- The open record marks when open0 is executed on the host or on a node processor. If it is generated on a node, then it records only the time-stamp and the node ID. If it is generated on the host, then it also indicates the number of processors that were allocated by the host. Examples:

```
open  clock 0 329 node 24
open  clock 0 1000 node -32768 allocating 64 processors
```

- The `load` record marks when `load0` is executed on the host. It also records the value of the node parameter passed to `load0`, which indicates which node processor was loaded. A value of "-1" for node means that all allocated processors were loaded by this call. Example:

```
load  clock 0 2000 node -32768 loading node 23
```

- The send record marks when send0 is executed on the host or on a node. It records the message destination, the message type, and the length of the message (in bytes). Example:

```
send  clock 2 524347 node 12 to -32768 type 8 lth 256
```

- The `recv` record marks when `recv0` is executed on the host or on a node and *the message that it is seeking is immediately found.* It records the message source, the message type, and the length of the message (in bytes). Example:

```
recv  clock 24 772354 node 63 from 0 type 16 lth 329
```

- The **recv_blocking** record marks when **recv0** is executed on the host or on a node and *the message that it is seeking is not immediately found.* It records the message type being sought. If this value is -1, then any message will satisfy the request. Example:

```
recv_blocking  clock 14 897000 node -32768 type 0
```

- The **recv_waking** record marks when a **recv0** call (executed on the host or on a node) that has been blocked finally receives the message it has been waiting for. The time-stamp records when the program exits from **recv0**. It records the message source, the message type, and the length of the message (in bytes). Example:

```
recv_waking  clock 15 145000 node -32768 from 0 type 0 lth 420
```

- The **message** record marks when **message0** is executed on the host or on a node. Example:

```
message  clock 0 986454 node 56
```

- The **sync** record marks when **sync0** is executed on a node. Note that it only indicates when the routine is entered, not when it is exited. Example:

```
sync  clock 429 616 node 15
```

- The **close** record marks when **close0** is executed on the host or on a node. Example:

```
close  clock 13 534666 node 3
```

- The **trace_level** record marks when **tracelevel** is executed on the host or on a node. It records the new values of the tracing level parameters (set by this call). Example:

```
trace_level  clock 3 11 node 6 event 3 compstats 3 commstats 3
```

- The **trace_mark** record marks when **tracemark** is executed on the host or on a node. It records the user-specified type associated with this call to tracemark. Example:

```
trace_mark  clock 3215 465799 node 22 type 17
```

- The **trace_stop** record marks when the trace buffer fills up and automatic flushing is *turned off* (on the host or a node). (See the manual pages for **tracehost** and **tracenode** for a description of this option.) In this case, no trace records are generated after the trace buffer fills up until either a **traceflush** is executed or the program ends. (A few trace records are always generated near the end of a program. The trace buffer is managed so that there is always enough space for these final records.) Example:

```
trace_stop  clock 15 333453 node 2 : ran out of memory
```

- The **trace_flush** record marks a call to **traceflush** on the host or node. Note that **traceflush** can be called either explicitly by an application program or implicitly when a trace buffer fills up and the automatic flushing is *turned on*. (See the manual pages for **tracehost** and **tracenode** for a description of this option.) Since flushing a trace buffer can significantly perturb the execution of a program, this record also marks when the program exits **traceflush**. Example:

```
trace_flush  clock 12 990311 node 77 finished 13 6645
```

- The **trace_exit** record marks a call to **traceexit** on the host or node. It also records the *total* number of bytes of trace data generated, which is useful in adjusting the amount of storage allocated for tracing in subsequent runs of a program. Note that **traceexit** is called automatically within close0 *on a node processor* if it isn't called explicitly in the node program. This is part of the logic that ensures that all trace data is sent to the host before interprocessor communication is disabled. Example:

```
trace_exit  clock 25 442678 node 8 space 145024
```

- The `block_begin` record marks the entry into a high-level communication routine executed on the host or on a node. It records a *block type*, normally used to identify the routine being called, a *location type*, normally used to identify which call to the routine is being recorded, and a *parameter type*, normally used to identify some attribute of the parameter values used in this call. The current interpretation of these values for the high-level routines is described in Table 4. Note that all "global combining" routines use the same block type. The user will need to specify a distinct message type for each "global combining" routine in order to be able to distinguish between them in the trace file. The high-level routines are meant to be examples of *user routines* that are built using the PICL primitives. Any additions to the set of high-level routines will also generate `block_begin` (and `block_end`) records if they are modeled after the current routines, but the choice of values for *block type*, *location type*, and *parameter type* must be made by the programmers of the new routines. Example:

```
block_begin  clock 33 111 node 8 block_type 3 location_type 41
 parameter_type 0
```

- The `block_end` record marks the exit from a high-level communication routine executed on the host or on a node. Its parameters and their interpretations are the same as those of a `block_begin` record. Example:

```
block_end  clock 33 844 node 8 block_type 3 location_type 41
 parameter_type 0
```

6.2. Computation statistics record

There is a single type of trace record for recording computation statistics. This `compstats` record has two fields in addition to the usual time-stamp and processor ID: *cumulative busy time* and *cumulative idle time*. The correct interpretation of busy and idle times is a matter of semantics. For example, idle time is sometimes defined to be the time during which a program is blocked inside `recv0`, and this can be calculated from the event trace records. However, the busy/idle times recorded in a `compstats`

routine	block type	location type	parameter type
barrier0	-1	0	0
bcast0	-2	type	root
bcast1	-3	type	root
gand0	-4	msgtype	root
gcomb0	-4	msgtype	root
gmax0	-4	msgtype	root
gmin0	-4	msgtype	root
gor0	-4	msgtype	root
gprod0	-4	msgtype	root
gsum0	-4	msgtype	root
gxor0	-4	msgtype	root

Table 4: **blockbegin** and **blockend** field values for high-level routines.

record are defined as follows: *A processor is idle if it is executing system routines that would not be necesary if executed on a serial computer; otherwise, it is busy.* By this definition, idle time is all of the time spent in a call to recv0, not just the time spent blocked waiting for a message. It also includes all of the time spent in open0, load0, and send0, if on the host or on a node, in message0 and sync0 if on a node, and in close0 if on the host. The "highest level" of tracing generates a compstats record whenever the idle time is modified, which is precisely whenever one of the routines listed above is called. In this case, the routine generates two compstats records, one immediately before the system call and one immediately after. In the first record, only the cumulative busy time changes with respect to the previous compstats record, while, in the second record, the idle time also changes. Depending on the level of tracing, a compstats record may also be generated in tracemark, at the beginning and end of a high-level communication routine, and at the end of the program. The compstats parameter to tracelevel is used to specify exactly when compstats records are generated. See the manual pages for tracelevel for details on how to use this parameter to control the recording of computation statistics in the trace file. Example:

```
compstats  clock 4 66978 node 23 busy 2 997566 idle 1 69412
```

6.3. Communication statistics record

There is a single type of trace record for recording communication statistics. The commstats record has five fields in addition to the usual time-stamp and processor

ID: *cumulative number of messages received, cumulative number of bytes in messages received, cumulative number of messages sent, cumulative number of bytes in messages sent,* and *cumulative number of probes of the message queue.* The last field is important because the time required to probe the message queue is usually too short to measure accurately by simply calling the system clock before and after the routine. As a result, the time spent probing the message queue is not recorded as idle time, and, cumulatively, can represent a significant amount of "hidden" idle time. The cumulative number of probes can be used to determine whether the recorded idle time is significantly underestimated. The "highest level" of tracing generates a `commstats` record whenever any of the fields is modified, which is whenever `send0`, `recv0`, `sync0`, or `message0` is called. For this case, two `commstats` records are generated, one immediately before the system call and one immediately after. Thus, the first record is exactly the same as the previous `commstats` record except for the time-stamp. In the second record, all of the fields are updated. Depending on the level of tracing, a `compstats` record may also be generated in `tracemark`, at the beginning and end of a high-level communication routine, and at the end of the program. The `commstats` parameter to `tracelevel` is used to specify exactly when `commstats` records are generated. See the manual pages for `tracelevel` for details on how to use this parameter to control the recording of communication statistics in the trace file. Example:

```
commstats  clock 18 445645 node 1 received 202 volume 20556 sent 109
 volume 4056 probed 39
```

6.4. Trace message record

The final type of trace record is that produced by a call to `tracemsg` on the host or the node. The routine `tracemsg` sends a string of length no more than 80 bytes that is placed directly into the trace file as soon as it is received by the host. The `trace_message` record has three fields: time-stamp, node ID, and *character string.* The time-stamp reflects when `tracemsg` was called. Example:

```
trace_message  clock 0 514332 node 10 This is a test of tracemsg.
```

6.5. Compact trace record format

Each compact trace record begins with an integer that identifies the type of trace record. Each of these integers corresponds to one of the keywords used in the verbose format to identify the record type, as indicated in Table 5.[1]

compact format integer	verbose format keyword
1	trace_start
2	open
3	load
4	send
6	recv
7	recv_blocking
8	recv_waking
9	message
10	sync
11	compstats
12	commstats
13	close
14	trace_level
15	trace_mark
16	trace_message
17	trace_stop
18	trace_flush
19	trace_exit
20	block_begin
21	block_end

Table 5: Verbose and compact format trace record type identifiers.

In general, to produce a compact trace record from a verbose trace record, replace the "type" keyword by the integer value described above, and then remove all other keywords. To produce a verbose trace record from a compact trace record, replace the "type" integer by the keyword described above, and insert keywords before (and possibly after) the integer data fields as indicated by the description of the appropriate type in Sections 6.1, 6.2, 6.3, and 6.4.

The one exception to this rule is that the compact compstats record does not explicitly record the cumulative busy time. It contains, in order, an integer indicating the record type, two integers specifying the time-stamp, an integer specifying the pro-

[1]Note that the number 5 is not currently used as a compact format record type. It has been reserved for a probe record type in case we are ever able to instrument probe0 unobtrusively.

cessor ID number, and two integers specifying the cumulative busy time. The following example corresponds to the verbose **compstats** example in Section 6.2:

```
11 4 66978 23 1 69412
```

To calculate the cumulative busy time, subtract the cumulative idle time from the time-stamp. For this example, the cumulative busy time is `2 997566`.

6.6. Order of trace records

The trace file will usually contain trace records that are sorted first by node ID number, and then by time-stamp, but this is not guaranteed. For example, **trace_message** records are placed in the trace file as soon as **tracemsg** is called, modulo the time it takes for the message to reach the host and the time it takes for the host to process it. Most of the other trace records are not received by the host until the end of all node programs. Moreover, if **traceflush** is used, then almost any order of the trace records is possible. If desired, the trace records can easily be (re)sorted by post-processing the trace file.

7. Machine dependencies

This section describes the machine-dependent aspects of PICL. For each machine, we first describe the valid range of values for each machine-dependent input parameter. Second, we document the additional execution time incurred when using PICL routines. Third, we describe how to link the library with a user-application code. This part of the documentation will receive periodic updates when more target machines are added, and when changes in the implementation of the library affect the overhead of using the routines.

7.1. iPSC/2

7.1.1. Legal parameter values

In this section, we indicate what values for each machine-dependent PICL input parameter are valid on the iPSC/2. We also describe any machine-dependent interpretations of the parameter values. If a parameter is not mentioned in the following list, then it is not machine-dependent, and its range of valid values should be clear from its description earlier in this manual.

An Intel iPSC/2 multiprocessor can have no more than 128 node processors, but the maximum number of processors at a particular site is installation-dependent. We use the variable P to refer to the number of processors allocated to a user.

The default integer size is 32 bits, and the possible values span the range from -2,147,483,648 to 2,147,483,647.

open0

- The node version of `open0` has no input parameters. The host version of `open0` has one input parameter, `numproc`. Valid values for `numproc` are

$$-2,147,583,648 \leq \texttt{numproc} \leq 128.$$

  If `numproc` is positive, then `open0` will attempt to allocate that many processors. If `numproc` is less than or equal to zero, then `open0` will attempt to use a previously allocated set of processors.

probe0

- `probe0` has one parameter, `type`. Valid values for `type` are

$$-1 \leq \texttt{type} < 900,000,000.$$

recv0

- **recv0** has three parameters: **buf**, **bytes**, and **type**. **buf** must be a legal memory address and **bytes** must be a valid positive integer. Valid values for **type** are

$$-1 \leq \texttt{type} < 900,000,000.$$

send0

- **send0** has four parameters: **buf**, **bytes**, **type**, and **dest**. **buf** must be a legal memory address. When sending to a node processor from a node processor, **bytes** must be a valid positive integer. When sending to the host or from the host, valid values for **bytes** are

$$1 \leq \texttt{bytes} \leq 256,000.$$

Valid values for **type** are

$$0 \leq \texttt{type} < 900,000,000.$$

Valid values for **dest** are

$$(-1 \leq \texttt{dest} \leq P-1) \text{ or } \texttt{dest} = -32,768.$$

If **dest** is nonnegative, then **send0** sends the message to the indicated node processor. If **dest** is -1, then **send0** sends the message to all allocated node processors. If **dest** is -32,768, then **send0** sends the message to the host.

7.1.2. Other machine dependencies

- If **load0** is used to load a process onto a node processor on which there is already an active process, then the original process on the processor is killed, and the new process begins execution normally.

- The routines **barrier0** and **setarc0** of the high-level PICL routines use default machine-dependent message types. Since the high-level routines are meant to represent typical user extensions to the library, these message types are within

the range permitted to any user application code. In consequence, the user has the ability to write seemingly correct code that will still execute incorrectly. For more information on what to watch out for, see the documentation for `barrier0` and `setarc0`. To be safe when using the high-level routines, a user application program should use message types that are strictly smaller than those used by `barrier0` and `setarc0`. On the iPSC/2, use message types that are *smaller* than 800,000,000.

- The resolution of the default iPSC/2 system clock on the node processors is 1 millisecond, which is too coarse to resolve the ordering of many important events in an application code. But, if `traceenable` is called *before* `load0` is used to load the node programs, then a high-resolution clock routine is used that returns the time in microseconds. Otherwise, the default system clock is used.

  Using the high-resolution clock routine has the side-effect of turning off memory protection in the node operating system. This may be unacceptable when debugging a node program. If so, simply call `traceenable` *after* `load0` to leave the memory protection unimpaired.

7.1.3. Performance

When the tracing logic is not activated, there is very little overhead incurred by using the PICL routines *on the node processors.* The very possibility of tracing causes `recv0` on the host to be 2-3 times slower than the native commands. This is the only major difference in the PICL overhead between the host and the node processors when not tracing. Since the host tends to be significantly slower than the nodes even when not using PICL, most application codes minimize the use of the host. In our experience, the additional overhead on the host is not important.

If the tracing logic is enabled and the trace information is sent to the trace file at arbitrary points in the node and host programs, then the additional cost due to tracing can be extremely high. But, if trace information is sent back only at the end of the node processes, then the cost is reasonable. The tables below provide empirical measures of this cost on a 64 processor iPSC/2.

Send/receive costs. Table 6 gives the time required to send one byte of data to an immediate neighbor and receive another byte back. Thus, this time represents the minimum cost associated with exchanging information. The table gives the time for this operation using the native commands `csend` and `crecv` (iPSC/2) and using the PICL routines `send0` and `recv0` with tracing off (notrace), with tracing on but no trace records generated (t000), with tracing on and "event" trace records generated (t300), with tracing on and "compstats" trace records generated (t030), with tracing on and "commstats" trace records generated (t003), and with tracing on and all possible trace records generated (t333), respectively.

iPSC/2	notrace	t000	t300	t030	t003	t333
480.0	500.0	827.7	944.4	864.2	875.9	1032.6

Table 6: Time spent exchanging one byte messages between neighboring nodes on the iPSC/2 (in microseconds).

Simply using the PICL routines increases the execution time very little, while tracing increases the execution time anywhere from 74 percent to 118 percent. Note that the tracing overhead is independent of message length. Thus, when larger messages are exchanged, the percentage overhead will decrease. For example, Table 7 gives the time required to exchange 1000 byte messages between neighboring processors.

iPSC/2	notrace	t000	t300	t030	t003	t333
1460.0	1480.0	1820.0	1951.4	1859.3	1885.1	2060.7

Table 7: Time spent exchanging 1000 byte messages between neighboring nodes on the iPSC/2 (in microseconds).

Similarly, the percentage overhead will decrease if the processors are not synchronized and one of them is "blocked" waiting for the message.

Basic PICL node routine costs. Table 8 contains the minimum times required to execute the low-level and tracing PICL routines on a node processor, as compared with the minimum time required to execute the corresponding native commands. The first column of execution times contains the times required when calling the native commands (iPSC/2). The other columns contain the times required to call the PICL routines with tracing off (notrace), with tracing on but only the minimum number of trace records generated (t000), with tracing on and "event" trace records generated

(t300), with tracing on and "compstats" trace records generated (t030), with tracing on and "commstats" trace records generated (t003), and with tracing on and all possible trace records generated (t333), respectively. An iPSC/2 time is marked as na if there is no native iPSC/2 command corresponding to the specified function.

function	iPSC/2	notrace	t000	t300	t030	t003	t333
open	na	50.0	268.0	275.0	280.0	290.0	305.0
clock	44.0	124.0	124.0	124.0	124.0	124.0	124.0
probe	40.0	41.9	41.9	41.9	41.9	41.9	41.9
recvinfo	90.0	92.0	92.0	92.0	92.0	92.0	92.0
message	35000.0	1400.0	1418.0	1446.0	1494.0	1537.0	1555.0
who	18.0	6.6	6.6	6.6	6.6	6.6	6.6
tracenode	na	na	335.0	335.0	335.0	335.0	335.0
tracelevel	na	20.0	87.0	87.0	99.7	103.1	123.1
traceinfo	na	10.0	14.5	14.5	14.5	14.5	14.5
tracemark	na	5.0	70.0	81.5	83.2	86.7	105.1
tracemsg	23000.0	5.0	770.0	770.0	770.0	770.0	770.0
traceflush	na	5.0	376.0	376.0	376.0	376.0	376.0
traceexit	na	5.0	98.0	98.0	98.0	98.0	98.0

Table 8: Cost of machine-dependent and machine-independent primitives on the iPSC/2 nodes (in microseconds).

Note that the iPSC/2 `tracemsg` time refers to the time required to use `fprintf` to write directly to the trace file. Note also that the execution times for any function that communicates with the host, like `tracemsg`, `traceflush`, and `message`, can vary wildly. All measurements reported here are the smallest times observed over a small number of runs.

`sync0` costs. The `sync0` routine attempts to synchronize all of the node processors in the sense that all node programs exit the routine at approximately the same time. The sharpness of the synchronization depends somewhat on when, and in what order, the processors call the routine, but we have found our algorithm to be surprisingly effective. This is very important since `sync0` is used to normalize time-stamps in trace records generated on different node processors when the `sync` option is specified in `tracenode`. We have yet to observe trace records normalized in this way whose time stamps violate the true partial order of events in an application code, for example, by messages appearing to be received before they are sent.

A *very* conservative upper bound on the difference between the time the first processor exits the routine and the last processor exits the routine is the time a processor spends in sync0 when the processors are already synchronized. This time is reported in Table 9 as a function of the number of processors. The first column of execution times contains the times required when calling the native command gsync (iPSC/2). The other columns contain the times required to call sync0 with tracing off (notrace), with tracing on but only the minimum number of trace records generated (t000), with tracing on and "event" trace records generated (t300), with tracing on and "compstats" trace records generated (t030), with tracing on and "commstats" trace records generated (t003), and with tracing on and all possible trace records generated (t333), respectively.

processors	iPSC/2	notrace	t000	t300	t030	t003	t333
2	474.9	501.8	647.0	656.0	668.3	687.1	706.6
4	949.9	983.8	1135.9	1142.7	1147.0	1171.8	1194.6
8	1495.0	1477.9	1623.2	1634.8	1641.9	1649.2	1688.4
16	2089.1	1964.0	2115.6	2120.1	2135.9	2151.9	2180.7
32	2658.6	2476.5	2617.5	2629.4	2648.6	2667.3	2683.1
64	3190.0	2960.0	3114.1	3128.2	3141.2	3174.6	3188.9

Table 9: Cost of machine-dependent and machine-independent synchronization primitives on the iPSC/2 nodes (in microseconds).

Note that sync0 does not call the native synchronization command gsync, and it is faster than gsync when eight or more processors are used.

7.1.4. Compiling and linking

PICL comes in two parts, a host library and a node library. We will refer to the host library as `hostlib.a` and to the node library as `nodelib.a`.

`hostlib.a` should be linked in whenever compiling a host program. For example, to compile a host program called `hostprogram.c` and link in the library routines use

```
cc -o hostprogram hostprogram.c hostlib.a -host
```

Similarly, to compile a host program called `hostprogram.f` and link in the library routines use

```
f77 -o hostprogram hostprogram.f hostlib.a -host
```

nodelib.a should be linked in whenever compiling a node program. For example, to compile a node program called **nodeprogram.c** and link in the library routines use

```
cc -o nodeprogram nodeprogram.c nodelib.a -node
```

Similarly, to compile a node program called **nodeprogram.f** and link in the library routines use

```
f77 -o nodeprogram nodeprogram.f nodelib.a -node
```

Note that a user may wish to use compiler and loader switches that are not indicated in the above examples.

7.2. iPSC/860

7.2.1. Legal parameter values

In this section, we indicate what values for each machine-dependent PICL input parameter are valid on the iPSC/860. We also describe any machine-dependent interpretations of the parameter values. If a parameter is not mentioned in the following list, then it is not machine-dependent, and its range of valid values should be clear from its description earlier in this manual.

An Intel iPSC/860 multiprocessor can have no more than 128 node processors, but the maximum number of processors at a particular site is installation-dependent. We use the variable P to refer to the number of processors allocated to a user.

The default integer size is 32 bits, and the possible values span the range from -2,147,483,648 to 2,147,483,647.

open0

- The node version of `open0` has no input parameters. The host version of `open0` has one input parameter, `numproc`. Valid values for `numproc` are

$$-2,147,583,648 \leq \texttt{numproc} \leq 128.$$

  If `numproc` is positive, then `open0` will attempt to allocate that many processors. If `numproc` is less than or equal to zero, then `open0` will attempt to use a previously allocated set of processors.

probe0

- `probe0` has one parameter, `type`. Valid values for `type` are

$$-1 \leq \texttt{type} < 900,000,000.$$

recv0

- `recv0` has three parameters: `buf`, `bytes`, and `type`. `buf` must be a legal memory address and `bytes` must be a valid positive integer. Valid values for `type` are

$$-1 \leq \texttt{type} < 900,000,000.$$

send0

- `send0` has four parameters: `buf`, `bytes`, `type`, and `dest`. `buf` must be a legal memory address. When sending to a node processor from a node processor, `bytes` must be a valid positive integer. When sending to the host or from the host, valid values for `bytes` are

$$1 \leq \texttt{bytes} \leq 256,000.$$

  Valid values for `type` are

$$0 \leq \texttt{type} < 900,000,000.$$

  Valid values for `dest` are

$$(-1 \leq \texttt{dest} \leq P-1) \text{ or } \texttt{dest} = -32,768.$$

  If `dest` is nonnegative, then `send0` sends the message to the indicated node processor. If `dest` is -1, then `send0` sends the message to all allocated node processors. If `dest` is -32,768, then `send0` sends the message to the host.

7.2.2. Other machine dependencies

- If `load0` is used to load a process onto a node processor on which there is already an active process, then the original process on the processor is killed, and the new process begins execution normally.

- The routines `barrier0` and `setarc0` of the high-level PICL routines use default machine-dependent message types. Since the high-level routines are meant to represent typical user extensions to the library, these message types are within

the range permitted to any user application code. In consequence, the user has the ability to write seemingly correct code that will still execute incorrectly. For more information on what to watch out for, see the documentation for `barrier0` and `setarc0`. To be safe when using the high-level routines, a user application program should use message types that are strictly smaller than those used by `barrier0` and `setarc0`. On the iPSC/860, use message types that are *smaller* than 800,000,000.

7.2.3. Performance

When the tracing logic is not activated, there is very little overhead incurred by using the PICL routines *on the node processors.* The very possibility of tracing causes `recv0` on the host to be 2-3 times slower than the native commands. This is the only major difference in the PICL overhead between the host and the node processors when not tracing. Since the host tends to be significantly slower than the nodes even when not using PICL, most application codes minimize the use of the host. In our experience, the additional overhead on the host is not important.

If the tracing logic is enabled and the trace information is sent to the trace file at arbitrary points in the node and host programs, then the additional cost due to tracing can be extremely high. But, if trace information is sent back only at the end of the node processes, then the cost is reasonable. The tables below provide empirical measures of this cost on a 128 processor iPSC/860 on which the node processors have a 40 Mhz clock.

Send/receive costs. Table 10 gives the time required to send one byte of data to an immediate neighbor and receive another byte back. Thus, this time represents the minimum cost associated with exchanging information. The table gives the time for this operation using the native commands `csend` and `crecv` (iPSC/860) and using the PICL routines `send0` and `recv0` with tracing off (notrace), with tracing on but no trace records generated (t000), with tracing on and "event" trace records generated (t300), with tracing on and "compstats" trace records generated (t030), with tracing on and "commstats" trace records generated (t003), and with tracing on and all possible trace records generated (t333), respectively.

iPSC/860	notrace	t000	t300	t030	t003	t333
99.7	113.5	141.5	171.6	161.1	172.9	222.9

Table 10: Time spent exchanging one byte messages between neighboring nodes on the iPSC/860 (in microseconds).

Simply using the PICL routines increases the execution time very little, while tracing increases the execution time anywhere from 42 percent to 124 percent. Note that the tracing overhead is independent of message length. Thus, when larger messages are exchanged, the percentage overhead will decrease. For example, Table 11 gives the time required to exchange 1000 byte messages between neighboring processors. Note that the absolute overhead also decreases for this case, reflecting the overlap of the overhead with the time spent in interprocessor communication.

iPSC/860	notrace	t000	t300	t030	t003	t333
807.9	814.4	836.6	847.1	846.7	855.4	876.4

Table 11: Time spent exchanging 1000 bytes messages between neighboring nodes on the iPSC/860 (in microseconds).

Similarly, the percentage overhead will decrease if the processors are not synchronized and one of them is "blocked" waiting for the message.

Basic PICL node routine costs. Table 12 describes the minimum times required to execute the low-level and tracing PICL routines on a node processor, as compared with the minimum time required to execute the corresponding native commands. The first column of execution times contains the times required when calling the native commands (iPSC/860). The other columns contain the times required to call the PICL routines with tracing off (notrace), with tracing on but only the minimum number of trace records generated (t000), with tracing on and "event" trace records generated (t300), with tracing on and "compstats" trace records generated (t030), with tracing on and "commstats" trace records generated (t003), and with tracing on and all possible trace records generated (t333), respectively. An iPSC/860 time is marked as na if there is no native iPSC/860 command corresponding to the specified function.

Note that the iPSC/860 `tracemsg` time refers to the time required to use `fprintf` to write directly to the trace file. Also note that the execution times for any function that communicates with the host, like `tracemsg`, `traceflush`, and `message`, can vary

function	iPSC/860	notrace	t000	t300	t030	t003	t333
open	na	36.7	77.9	80.4	90.8	86.1	96.5
clock	2.8	3.0	3.0	3.0	3.0	3.0	3.0
probe	5.0	5.5	5.5	5.5	5.5	5.5	5.5
recvinfo	2.6	5.3	5.3	5.3	5.3	5.3	5.3
message	30571.4	217.6	235.1	239.9	247.9	253.3	255.0
who	2.7	0.8	0.8	0.8	0.8	0.8	0.8
tracenode	na	na	27.5	27.5	27.5	27.5	27.5
tracelevel	na	1.2	4.7	5.5	6.6	7.6	9.4
traceinfo	na	1.1	1.4	1.4	1.4	1.4	1.4
tracemark	na	0.6	2.2	3.7	4.8	5.1	9.2
tracemsg	14419.6	0.6	170.3	165.0	160.9	162.1	159.7
traceflush	na	0.6	70.2	68.8	71.4	69.3	74.5
traceexit	na	0.6	7.7	7.7	7.7	7.7	7.7

Table 12: Cost of machine-dependent and machine-independent primitives on the iPSC/860 nodes (in microseconds).

wildly. All measurements reported here are the smallest times observed over a small number of runs.

sync0 costs. The sync0 routine attempts to synchronize all of the node processors in the sense that all node programs exit the routine at approximately the same time. The sharpness of the synchronization depends somewhat on when, and in what order, the processors call the routine, but we have found our algorithm to be surprisingly effective. This is very important since sync0 is used to normalize time-stamps in trace records generated on different node processors when the sync option is specified in tracenode. We have yet to observe trace records normalized in this way whose time stamps violate the true partial order of events in an application code, for example, by messages appearing to be received before they are sent.

A *very* conservative upper bound on the difference between the time the first processor exits the routine and the last processor exits the routine is the time a processor spends in sync0 when the processors are already synchronized. This time is reported in Table 13 as a function of the number of processors. The first column of execution times contains the times required when calling the native command gsync (iPSC/860). The other columns contain the times required to call sync0 with tracing off (notrace), with tracing on but only the minimum number of trace records generated (t000), with tracing on and "event" trace records generated (t300), with tracing on and "comp-

stats" trace records generated (t030), with tracing on and "commstats" trace records generated (t003), and with tracing on and all possible trace records generated (t333), respectively.

processors	iPSC/860	notrace	t000	t300	t030	t003	t333
2	107.7	115.6	127.7	131.3	139.4	143.5	158.6
4	218.6	218.3	232.2	238.5	242.6	247.9	263.8
8	361.0	331.3	350.7	350.2	362.1	364.6	384.1
16	521.7	449.3	464.0	467.2	478.7	480.6	498.0
32	666.3	559.9	582.8	584.6	591.2	601.8	618.2
64	809.4	681.5	696.9	704.7	708.6	716.2	735.6
128	941.9	763.1	786.7	790.4	804.5	802.7	824.4

Table 13: Cost of machine-dependent and machine-independent synchronization primitives on the iPSC/860 nodes (in microseconds).

Note that `sync0` does not call the native synchronization command `gsync`, and it is faster than `gsync` when four or more processors are used.

7.2.4. Compiling and linking

PICL comes in two parts, a host library and a node library. We will refer to the host library as `hostlib.a` and to the node library as `nodelib.a`.

`hostlib.a` should be linked in whenever compiling a host program. For example, to compile a host program called `hostprogram.c` and link in the library routines use

```
cc -o hostprogram hostprogram.c hostlib.a -host
```

Similarly, to compile a host program called `hostprogram.f` and link in the library routines use

```
f77 -o hostprogram hostprogram.f hostlib.a -host
```

`nodelib.a` should be linked in whenever compiling a node program. For example, to compile a node program called `nodeprogram.c` and link in the library routines use

```
cc -i860 -o nodeprogram nodeprogram.c nodelib.a -node
```

Similarly, to compile a node program called `nodeprogram.f` and link in the library routines use

```
f77 -i860 -o nodeprogram nodeprogram.f nodelib.a -node
```

Note that a user may wish to use compiler and loader switches that are not indicated in the above examples.

7.3. Ncube/3200

7.3.1. Legal parameter values

In this section, we indicate what values for each machine-dependent PICL input parameter are valid on the Ncube/3200 (previously known as the Ncube/ten). We also describe any machine-dependent interpretations of the parameter values. If a parameter is not mentioned in the following list, then it is not machine-dependent, and its range of valid values should be clear from its description earlier in this manual.

An Ncube/3200 multiprocessor can have no more than 1024 node processors, but the maximum number of processors at a particular site is installation-dependent. We use the variable P to refer to the number of processors allocated to a user.

The default integer size on the node processors is 32 bits, and the possible values span the range from -2,147,483,648 to 2,147,483,647. The default integer size on the host processor is 16 bits, and the possible values span the range from -32,768 to 32,767.

open0

- The node version of `open0` has no input parameters. The host version of `open0` has one input parameter, `numproc`. Valid values for `numproc` are

$$-32,768 \leq \texttt{numproc} \leq 1024.$$

  If `numproc` is positive, then `open0` will attempt to allocate that many processors. If `numproc` is less than or equal to zero, then `open0` will allocate the maximum number of processors currently available.

probe0

- `probe0` has one parameter, `type`. Valid values for `type` are

$$-1 \leq \texttt{type} < 32,000.$$

recv0

- recv0 has three parameters: buf, bytes, and type. buf must be a legal memory address and bytes must be a valid positive integer. Valid values for type are

$$-1 \leq \texttt{type} < 32,000.$$

send0

- send0 has four parameters: buf, bytes, type, and dest. buf must be a legal memory address. When sending from a node, valid values for bytes are

$$0 \leq \texttt{bytes} \leq 27,000.$$

When sending from the host, valid values for bytes are

$$0 \leq \texttt{bytes} \leq 65,527.$$

Valid values for type are

$$0 \leq \texttt{type} < 32,000.$$

Valid values for dest are

$$(0 \leq \texttt{dest} \leq P-1) \text{ or } \texttt{dest} = -32,768.$$

If dest is nonnegative, then send0 sends the message to the indicated node processor. If dest is -32,768, then send0 sends the message to the host.

7.3.2. Other machine dependencies

- If load0 is used to load a process onto a node processor on which there is already an active process, then the original process on the processor is killed, and the new process begins execution normally.
- The routines barrier0 and setarc0 of the high-level PICL routines use default machine-dependent message types. Since the high-level routines are meant to

represent typical user extensions to the library, these message types are within the range permitted to any user application code. In consequence, the user has the ability to write seemingly correct code that will still execute incorrectly. For more information on what to watch out for, see the documentation for `barrier0` and `setarc0`. To be safe when using the high-level routines, a user application program should use message types that are strictly smaller than those used by `barrier0` and `setarc0`. On the Ncube/3200, use message types that are *smaller* than 30,000.

- The resolution of the Ncube/3200 system clock is a function of the hardware clock, which is installation-dependent. To correctly normalize the clock times produced by `clock0` and tracing for a given installation, it may be necessary to modify PICL header files and recompile the source. See Section 9 for a source of more detailed information.

- It is nontrivial to write C routines that can be called from FORTRAN on the Ncube/3200, and we will not support FORTRAN language calls to PICL routines until PICL has been implemented in FORTRAN.

7.3.3. Performance

When the tracing logic is not activated, there is very little overhead incurred by using the PICL routines *on the node processors*. The very possibility of tracing causes `recv0` on the host to be 2-3 times slower than the native commands. This is the only major difference in the PICL overhead between the host and the node processors when not tracing. In practice, the host tends to be slower than the nodes even when not using PICL, and most application codes minimize the use of the host. In our experience, the additional overhead on the host is not important.

If the tracing logic is enabled and the trace information is sent to the trace file at arbitrary points in the node and host programs, then the additional cost due to tracing can be extremely high. But, if trace information is sent back only at the end of the node processes, then the cost is reasonable. The tables below provide empirical measures of this cost on a 1024 processor Ncube/3200 on which the node processors have an 8 Mhz clock.

Send/receive costs. Table 14 gives the time required to send one byte of data to an immediate neighbor and receive another byte back. Thus, this time represents the minimum cost associated with exchanging information. The table gives the time for this operation using the native commands `nwrite` and `nread` (Ncube) and using the PICL routines `send0` and `recv0` with tracing off (notrace), with tracing on but no trace records generated (t000), with tracing on and "event" trace records generated (t300), with tracing on and "compstats" trace records generated (t030), with tracing on and "commstats" trace records generated (t003), and with tracing on and all possible trace records generated (t333), respectively.

Ncube	notrace	t000	t300	t030	t003	t333
683.5	801.3	1003.5	1159.7	1136.6	1214.7	1496.3

Table 14: Time spent exchanging one byte messages between neighboring nodes on the Ncube/3200 (in microseconds).

Simply using the PICL routines increases the execution time by approximately 17 percent, representing the cost of two extra levels of subroutine calls per `send0` and `recv0` call. Tracing increases the execution time anywhere from 47 percent to 119 percent. Note that the tracing overhead is independent of message length. Thus, when larger messages are exchanged, the percentage overhead will decrease. For example, Table 15 gives the time required to exchange 1000 byte messages between neighboring processors.

Ncube	notrace	t000	t300	t030	t003	t333
2899.2	2987.5	3147.5	3248.6	3234.6	3284.5	3503.4

Table 15: Time spent exchanging 1000 byte messages between neighboring nodes on the Ncube/3200 (in microseconds).

Similarly, the percentage overhead will decrease if the processors are not synchronized and one of them is "blocked" waiting for the message.

Basic PICL node routine costs. Table 16 describes the minimum times required to execute the low-level and tracing PICL routines on a node processor, as compared with the minimum time required to execute the corresponding native commands. The first column of execution times contains the times required when calling the native commands (Ncube). The other columns contain the times required to call the PICL

routines with tracing off (notrace), with tracing on but only the minimum number of trace records generated (t000), with tracing on and "event" trace records generated (t300), with tracing on and "compstats" trace records generated (t030), with tracing on and "commstats" trace records generated (t003), and with tracing on and all possible trace records generated (t333), respectively. An Ncube time is marked as **na** if there is no native Ncube command corresponding to the specified function.

function	Ncube	notrace	t000	t300	t030	t003	t333
open	na	256.0	384.0	384.0	384.0	512.0	640.0
clock	34.0	63.1	63.1	63.1	63.1	63.1	63.1
probe	51.2	98.6	98.6	98.6	98.6	98.6	98.6
recvinfo	na	49.9	49.9	49.9	49.9	49.9	49.9
message	na	4352.0	4352.0	4480.0	4480.0	4608.0	4608.0
who	41.3	30.3	30.3	30.3	30.3	30.3	30.3
tracenode	na	na	409.6	409.6	409.60	409.6	409.6
tracelevel	na	34.6	141.4	141.4	177.9	195.8	234.2
traceinfo	na	47.0	47.0	47.0	47.0	47.0	47.0
tracemark	na	16.6	81.9	112.6	117.8	135.7	202.2
tracemsg	na	16.6	3200.0	3200.0	3200.0	3200.0	3200.0
traceflush	na	16.6	384.0	384.0	384.0	384.0	384.0
traceexit	na	16.6	184.3	184.3	184.3	184.3	184.3

Table 16: Cost of machine-dependent and machine-independent primitives on the Ncube/3200 nodes (in microseconds).

Note that the execution times for any function that communicates with the host, like **tracemsg**, **traceflush**, and **message**, can vary wildly. All measurements reported here are the smallest times observed over a small number of runs.

sync0 costs. The **sync0** routine attempts to synchronize all of the node processors in the sense that all node programs exit the routine at approximately the same time. The sharpness of the synchronization depends somewhat on when, and in what order, the processors call the routine, but we have found our algorithm to be surprisingly effective. This is very important since **sync0** is used to normalize time-stamps in trace records generated on different node processors when the **sync** option is specified in **tracenode**. We have yet to observe trace records normalized in this way whose time stamps violate the true partial order of events in an application code, for example, by messages appearing to be received before they are sent.

A *very* conservative upper bound on the difference between the time the first processor exits the routine and the last processor exits the routine is the time a processor spends in **sync0** when the processors are already synchronized. This time is reported in Table 17 as a function of the number of processors. The first column of execution times contains no times since there is no native synchronization primitive on the Ncube/3200. The other columns contain the times required to call **sync0** with tracing off (notrace), with tracing on but only the minimum number of trace records generated (t000), with tracing on and "event" trace records generated (t300), with tracing on and "compstats" trace records generated (t030), with tracing on and "commstats" trace records generated (t003), and with tracing on and all possible trace records generated (t333), respectively.

processors	Ncube	notrace	t000	t300	t030	t003	t333
2	na	769.3	880.6	907.5	945.9	980.5	1071.4
4	na	1466.9	1578.2	1600.0	1643.5	1684.5	1774.1
8	na	2170.9	2281.0	2307.8	2350.1	2387.2	2479.4
16	na	2872.3	2983.7	3010.6	3054.1	3092.5	3183.4
32	na	3575.0	3687.7	3712.0	3758.1	3793.9	3887.4
64	na	4277.8	4390.4	4414.7	4458.2	4495.4	4591.4
128	na	4977.9	5091.8	5117.4	5162.2	5198.1	5291.5
256	na	5680.6	5795.8	5818.9	5863.7	5900.8	5994.2
512	na	6380.8	6497.3	6520.3	6565.1	6602.2	6695.7
1024	na	7082.2	7197.4	7220.5	7265.3	7303.7	7395.8

Table 17: Cost of machine-dependent and machine-independent synchronization primitives on Ncube/3200 nodes (in microseconds).

7.3.4. Compiling and linking

PICL comes in two parts, a host library and a node library. We will refer to the host library as `hostlib.a` and to the node library as `nodelib.a`.

`hostlib.a` should be linked in whenever compiling a host program. For example, to compile a host program called `hostprogram.c` and link in the library routines use

```
ucc -K -khNH42 -c host.c
ld -r -g -x1 -o host.ltl /lib/main42.o host.o hostlib.a /lib/ucc42.lib
lc -e host host.ltl
```

nodelib.a should be linked in whenever compiling a node program. For example, to compile a node program called **nodeprogram.c** and link in the library routines use

```
ucc -K -khNN -c node.c
ldn -o node.ltl -b -r2 /lib/main.on node.on nodelib.a /lib/uccn.lib
 /lib/f77n.lib /lib/end.on
lcn -e node node.ltl
```

Note that a user may wish to use compiler and loader switches that are not indicated in the above examples.

8. Getting PICL

The source code for PICL is available from **netlib** [Don87]. The PICL source is currently 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
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
cuserguide.shar	latex source of the user guide for the C version of PICL
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. Unshar 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 this is done, do 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**.

9. Further Information

If your need additional information, or if you have suggestions or complaints, contact:

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

Dr. Worley can also be contacted via electronic mail at
worley@msr.epm.ornl.gov

10. References

[Don87] J. J. DONGARRA AND E. GROSSE, *Distribution of mathematical software via electronic mail*, Comm. ACM, 30 (1987), pp. 403–407.

[Dun86] T. H. DUNIGAN, *A message-passing multiprocessor simulator*, Tech. Report ORNL/TM-9966, Oak Ridge National Laboratory, Oak Ridge, TN, May 1986.

[Gei90] G. A. GEIST, M. T. HEATH, B. W. PEYTON, AND P. H. WORLEY, *A users' guide to* PICL*: a portable instrumented communication library*, Tech. Report ORNL/TM-11616, Oak Ridge National Laboratory, Oak Ridge, TN, August 1990.

11. Quick Reference List

PICL Low-Level Primitives

void **check0**(int checking)

double **clock0**()

void **close0**(int release)

void **load0**(char *file, int node)

void **message0**(char *message)

void **open0**(int *numproc, int *me, int *host)

int **probe0**(int type)

void **recv0**(char *buf, int bytes, int type)

void **recvinfo0**(int *bytes, int *type, int *source)

void **send0**(char *buf, int bytes, int type, int dest)

void **sync0**()

void **who0**(int *numproc, int *me, int *host)

PICL High-Level Communication Routines

void **barrier0**()

void **bcast0**(char *buf, int bytes, int type, int root)

void **bcast1**(char *buf, int bytes, int type, int root)

void **gand0**(char *buf, int items, int datatype, int msgtype, int root)

void **gcomb0**(char *buf, int items, int datatype, int msgtype, int root, void (*comb)())

void **getarc0**(int *nprocs, int *top, int *ord, int *dir)

int **ginv0**(int i)

void **gmax0**(char *buf, int items, int datatype, int msgtype, int root)

void **gmin0**(char *buf, int items, int datatype, int msgtype, int root)

void **gor0**(char *buf, int items, int datatype, int msgtype, int root)

void **gprod0**(char *buf, int items, int datatype, int msgtype, int root)

int **gray0**(int i)

void **gsum0**(char *buf, int items, int datatype, int msgtype, int root)

void **gxor0**(char *buf, int items, int datatype, int msgtype, int root)

void **setarc0**(int *nprocs, int *top, int *ord, int *dir)

PICL Execution Tracing Routines

void **traceenable**(char *tracefile, int verbose)

void **traceexit**()

void **traceflush**()

void **tracehost**(int tracesize, int flush)

void **traceinfo**(int remaining, int event, int compstats, int commstats)

void **tracelevel**(int event, int compstats, int commstats)

void **tracemark**(int marktype)

void **tracemsg**(char *message)

void **tracenode**(int tracesize, int flush, int sync)

INTERNAL DISTRIBUTION

1. B. R. Appleton
2. E. F. D'Azevedo
3. J. J. Dongarra
4. J. B. Drake
5. T. H. Dunigan
6. R. E. Flanery
7-11. G. A. Geist
12. L. Gray
13-14. R. F. Harbison
15-19. M. T. Heath
20. E. R. Jessup
21. M. R. Leuze
22-26. F. C. Maienschein
27. E. G. Ng
28. C. E. Oliver
29. G. Ostrouchov
30-34. B. W. Peyton
35. W. M. Post
36-40. S. A. Raby
41. C. H. Romine
42-46. R. C. Ward
47-51. P. H. Worley
52. J. J. Dorning (EPMD Advisory Committee)
53. R. M. Haralick (EPMD Advisory Committee)
54. J. E. Leiss (EPMD Advisory Committee)
55. M. F. Wheeler (EPMD Advisory Committee)
56. N. Moray (EPMD Advisory Committee)
57. Central Research Library
58. ORNL Patent Office
59. K-25 Plant Library
60. Y-12 Technical Library /Document Reference Station
61. Laboratory Records - RC
62-63. Laboratory Records Department

EXTERNAL DISTRIBUTION

64. Dr. Edward H. Barsis, Computer Science and Mathematics, P. O. Box 5800, Sandia National Laboratory, Albuquerque, NM 87185

65. Dr. John Cavallini, Acting Director, Scientific Computing Staff, Applied Mathematical Sciences, Office of Energy Research, U.S. Department of Energy, Washington, DC 20545

66. Ms. Helen Davis, Computer Science Department, Stanford University, Stanford, CA 94305

67. Dr. Yuefan Deng, Applied Mathematics Department, SUNY at Stony Brook, Stony Brook, NY 11794-3600

68. Professor Larry Dowdy, Computer Science Department, Vanderbilt University, Nashville, TN 37235

69. Professor Geoffrey C. Fox, Physics Department, MS 356-48, California Institute of Technology, Pasadena, CA 91125

70. Professor Dennis B. Gannon, Computer Science Department, Indiana University, Bloomington, IN 47401

71. Dr. Alan George, Vice President, Academic and Provost, Needles Hall, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1

72. Dr. Gene Golub, Computer Science Department, Stanford University, Stanford, CA 94305

73. Prof. John L. Gustafson, Ames Laboratory, 236 Wilhelm Hall, Iowa State University, Ames, IA 50011-3020

74. Dr. Philip J. Hatcher, Department of Computer Science, College of Engineering and Physical Sciences, Kingsbury Hall, Durham, NH 03824

75. Dr. Bo Kagstrom, Institute of Information Processing, University of Umea, 5-901 87 Umea, Sweden

76. Professor Malvyn Kalos, Courant Institute for Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012

77. Dr. Hans Kaper, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439

78. Dr. Alan H. Karp, IBM Scientific Center, 1530 Page Mill Road, Palo Alto, CA 94304

79. Dr. Kenneth Kennedy, Department of Computer Science, Rice University, P.O. Box 1892, Houston, TX 77001

80. Dr. Tom Kitchens, ER-7, Applied Mathematical Sciences, Scientific Computing Staff, Office of Energy Research, Office G-437 Germantown, Washington, DC 20545

81. Dr. Ted Lewis, Research Director, Oregon Advanced Computing Institute, 19500 SW Gibbs Dr. #101, Beaverton, OR 97006

82. Dr. James McGraw, Lawrence Livermore National Laboratory, L-306, P. O. Box 808, Livermore, CA 94550

83. Dr. John Meissen, Oregon Advanced Computing Institute, 19500 SW Gibbs Dr., Suite 110, Beaverton, OR 97006-6907

84. Professor James M. Ortega, Department of Applied Mathematics, Thornton Hall, University of Virginia, Charlottesville, VA 22901

85. Dr. Angela Quealy, Sverdrup Technology, Inc., 2001 Aerospace Parkway, Brook Park, OH 44142

86. Professor Daniel A. Reed, Computer Science Department, University of Illinois, Urbana, IL 61801

87. Dr. Joel Saltz, ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23665

88. Dr. Ahmed H. Sameh, Center for Supercomputing R&D, 1384 W. Springfield Avenue, University of Illinois, Urbana, IL 61801

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

90. Dr. Rick Stevens, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439

91. Dr. Andrew B. White, Los Alamos National Laboratory, P. O. Box 1663, MS-265, Los Alamos, NM 87545

92. 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

93-102. Office of Scientific & Technical Information, P. O. Box 62, Oak Ridge, TN 37831