ANL-93/23

Users Manual for the Chameleon Parallel Programming Tools

by W. Gropp and B. Smith

Mathematics and Computer Science Division
Mathematics and Computer Science Division
Mathematics and Computer Science Division

Argonne National Laboratory, Argonne, Illinois 60439
operated by The University of Chicago
for the United States Department of Energy under Contract W-31-109-Eng-38

Reproduced 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

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

Distribution Category:
Mathematics and
Computer Science (UC-405)

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, IL 60439-4801

ANL-93/23

Users Manual for the Chameleon Parallel Programming Tools

by

William Gropp
Mathematics and Computer Science Division

Barry Smith
Department of Mathematics
University of California, Los Angeles

June 1993

This work was supported in part by the Office of Scientific Computing, U.S. Department of Energy and in part by the Office of Naval Research under contract ONR N00014-90-J-1695.

Contents

Users Manual for the Chameleon Parallel Programming Tools

by

William Gropp

Barry Smith

Abstract

Message passing is a common method for writing programs for distributed-memory parallel computers. Unfortunately, the lack of a standard for message passing has hampered the construction of portable and efficient parallel programs. In an attempt to remedy this problem, a number of groups have developed their own message-passing systems, each with its own strengths and weaknesses. Chameleon is a second-generation system of this type. Rather than replacing these existing systems, Chameleon is meant to supplement them by providing a uniform way to access many of these systems. Chameleon's goals are to (a) be very lightweight (low overhead), (b) be highly portable, and (c) help standardize program startup and the use of emerging message-passing operations such as collective operations on subsets of processors. Chameleon also provides a way to port programs written using PICL or Intel NX message passing to other systems, including collections of workstations.

Chameleon is tracking the Message-Passing Interface (MPI) draft standard and will provide both an MPI implementation and an MPI transport layer. Chameleon provides support for heterogeneous computing by using p4 and PVM. Chameleon's support for homogeneous computing includes the portable libraries p4, PICL, and PVM and vendor-specific implementation for Intel NX, IBM EUI (SP-1), and Thinking Machines CMMD (CM-5). Support for Ncube and PVM 3.x is also under development.

Chapter 1

Introduction

Chameleon is a collection of routines that provide a hierarchy of models for parallel programming on distributed-memory parallel computers. These routines are intended to provide a consistent, easy-to-use model of message passing that enables access to all of the power of a distributed-memory computer. An important feature of Chameleon is that, in combination with packages such as p4 and PVM, code that uses Chameleon (including the program startup routines) needs *no* changes to run on a collection of workstations as well as on parallel supercomputers such as the Intel Paragon and Thinking Machines CM-5. Another feature is that a wide variety of debugging information (both for correctness and for performance) is made available. A production library that imposes no overhead is also provided (again requiring no change in the source code).

Chameleon's overhead is low enough that it was used as the message-passing system in an application that won a Gordon Bell Prize in 1992 [9]. This application, BlockSolve (a package for solving large, sparse, symmetric linear systems), is publicly available and, because it uses Chameleon, is portable to a wide variety of systems.

The Chameleon library is organized as a large collection of fairly simple routines rather than as a small collection of complex routines. It is not necessary to learn about or even be familiar with most of the routines, though as an application is developed, many of the routines may come in handy. Chameleon is part of the Portable, Extensible Tools for Scientific computing (PETSc) library, which provides manual pages, portable makefiles, and a variety of numerical support software.

The relationship of Chameleon to other systems is shown in Figure 1.1. Note that some systems, such as PICL and NX, appear as both inputs to and outputs from Chameleon. This means that programs written using Chameleon can run on systems that provide PICL or NX as the message-passing system *and* that many programs written using PICL or NX can run on any system that Chameleon runs on. This "reverse compatibility" is discussed in more detail in

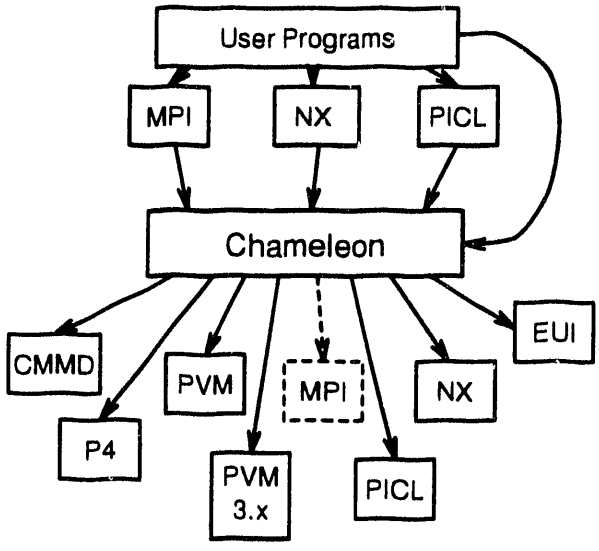

Figure 1.1: Relationship of Chameleon to other message-passing systems

Chapter 7.

1.1 Why Use Chameleon

Chameleon provides a standardized and extremely low overhead interface to message-passing software. It provides a uniform interface for program startup and simplifies the use of clusters of workstations or other computers (including massively parallel ones). It provides a stable interface to other packages that are continuing to change as they are developed. It provides routines for managing sets of processors (sometimes known as groups), including providing collective operations on these subsets of processors. Finally, Chameleon is part of a larger integrated package of routines (PETSc, for Portable, Extensible Tools for Scientific computing) that includes methods for solving large systems of linear and nonlinear equations, both sequentially and in parallel.

1.2 Organization of the Manual

This manual is organized roughly into four sections. Chapter 2 describes how programs are initialized in Chameleon. It also contains information on both homogeneous massively parallel processing (MPP) parallelism and heterogeneous, distributed-workstation parallelism. Chapter 3 describes point-to-point message passing routines. Chapter 4 describes the routines for collective operations, in-

cluding the routines for defining subsets of processes. The final chapters describe some special topics, such as debugging (Chapter 5) and examples (Chapter 8). The rest of this chapter describes various aspects of the Chameleon system; all users should at least skim this text. Section 1.8 explains how to write a `makefile` to use with Chameleon.

1.3 How to Read This Manual

This manual contains both simple examples and detailed information about using advanced functions. To make it easier for the novice, many of the chapters have a section titled "Getting Started." You can read just that section and skip the rest of the chapter. This will give you enough to get started with using Chameleon; you will need to read the rest of each chapter to discover all of Chameleon's functionality. In addition, you should read Chapter 5 on debugging.

This is not a reference manual; details of the routines are available through the man pages or the reference manual ('`tools.core/refs/refman.dvi`'). X Window system users may use '`tools.core/bin/toolman`' to read the man pages; this uses the `xman` program.

This manual does not discuss what message passing is or how to design programs using message passing.

1.4 Basic Routines

This section contains a short summary that will get you started with Chameleon. It does not contain all of the routines in all of their glory. The calling sequences for these routines can be found in Chapter 10; some example programs are presented in Chapter 8. Additional example programs may be found in '`tools.core/comm/examples`'.

Chameleon contains a large number of routines. The set described here is sufficient to write many portable message-passing programs. These routines are ordered roughly by use; all programs must use `PICall` to get started, most programs will use `PIbsend` and `PIbrecv` to send messages, and some programs will use `PIgdsum` to perform a global sum.

PICall	Call a routine in a parallel execution mode.
PImytid	Return my processor id.
PInumtids	Return the number of processors.
PIbrecv	Receive a message from another processor.
PIbsend	Send a message to another processor.
PIgXmin	Form the global minimum of a vector (**X** = **d** for double, **f** for float, etc.).

PIgXmax	Form the global maximum of a vector.
PIgXsum	Form the global sum of a vector.
PIbcast	Broadcast a vector to all other processors.

These will get you started. There are many additional routines, offering different kinds of sends and receives. There are additional global (or collective) operations; in addition, all global operations may be performed on subsets of processors. Various aids for debugging are also provided; see Chapter 5. Complete summaries of the calling sequences for these routines may be found in Chapter 10. There are man pages for all of these routines.

1.5 A Simple and Complete Example

The section presents a program, written in Fortran, for computing an approximation to the value of pi. This was chosen because it is a simple program that makes use of many of these routines. Two makefiles are also presented. One uses the PETSc makefiles to provide a portable makefile; the other is a makefile using the p4 system for a collection of Sun 4's. This second makefile, because it is more specific, may help in understanding the portable makefile.

This program is a modification of a pi program whose origins are unknown.

```
      integer funtion worker()
      integer           nprocs, myid, pinumtids, pimytid
      integer           INTSZE, MSG_INT, PSAllProcs
      integer           i, n
      double precision  pi, PI25DT, h, sum, x, f, a, temp
      parameter (INTSZE = 4, MSG_INT = 1, PSAllProcs = 0)
      parameter (PI25DT = 3.14159265358979323846264 3d0)

c --  function to intergrate
      f(a) = 4.d0 / (1.d0 + a*a)

      nprocs = PInumtids()
      myid   = PImytid()
10    if ( myid .eq. 0 ) then
         write(6,98)
98       format('Enter the number of intervals: (0 quits)')
         read(5,99)n
99       format(i10)
      endif
      Call PIbcastSrc(n, INTSZE, 0, PSAllProcs, MSG_INT)

c --  everyone check for quit signal
      if ( n .le. 0 ) goto 30
```

```
c --  calculate the interval size
      h    = 1.0d0/n
      sum  = 0.0d0
      do 20 i = myid+1, n, nprocs
         x   = h * (dble(i) - 0.5d0)
         sum = sum + f(x)
20    continue
      pi = h * sum

c --  collect all the partial sums
      call PIgdsum(pi, 1, temp, PSAllProcs)

c --  node 0 prints the answer.
      if (myid .eq. 0) then
         write(6, 97) pi, abs(pi - PI25DT)
      endif
      goto 10

97    format('  pi is approximately: ', F18.16,'  Error is: ', F18.16)
30    continue
      return
      end
```

This program asks for the number of intervals to use and sends that value to all the processors (with **PIbcastSrc**). Each processor then computes its contribution and uses **PIgdsum** to compute the sum of the contributions.

The makefile for this example is

```
ALL:  pi

ITOOLSDIR = /usr/local/tools.core

CFLAGS    = -I$(ITOOLSDIR) $(OPT) $(COPT)
LDIR      = $(ITOOLSDIR)/libs/libs$(BOPT)$(PROFILE)/$(ARCH)
LIBS      = $(LDIR)/tools$(COMM).a $(LDIR)/tools.a \
            $(LDIR)/tools$(COMM).a $(LDIR)/system.a
FLIBS     = $(ITOOLSDIR)/fort/$(ARCH)/fort$(COMM).a \
            $(ITOOLSDIR)/fort/$(ARCH)/fort.a

include $(ITOOLSDIR)/bmake/$(ARCH).$(COMM)
include $(ITOOLSDIR)/bmake/$(ARCH).$(BOPT)$(PROFILE)
include $(ITOOLSDIR)/bmake/$(ARCH)

pi: pi.o
        $(FLINKER) -o pi $(BASEOPTF) pi.o \
                   $(FLIBS) $(LIBS) $(CLIB) $(SLIB) -lm
```

This makefile relies on features of the PETSc system and requires a few values to be specified on the **make** line. For example, to build the program **pi** to run on a collection of Sun 4 machines using p4, use

```
make ARCH=sun4 COMM=p4 BOPT=g
```

The **BOPT=g** produces a version with extra debugging support. **BOPT=O** should be used for production runs.

The second makefile for this example does not require any additional makefiles; as such, it is specific to a particular platform. In this case, the makefile supports p4 on Sun 4's.

```
ALL:  pi

ITOOLSDIR = /usr/local/tools.core
LDIR      = $(ITOOLSDIR)/libs/libsO/sun4
LIBS      = $(LDIR)/toolsp4.a $(LDIR)/tools.a \
            $(LDIR)/toolsp4.a $(LDIR)/system.a
FLIBS     = $(ITOOLSDIR)/fort/sun4/fortp4.a \
            $(ITOOLSDIR)/fort/sun4/fort.a
pi: pi.o
        f77 -o pi -O pi.o $(FLIBS) $(LIBS) \
                  /usr/local/p4-1.2c/SUN/libp4.a -lm
```

Note that this makefile also selects a particular optimization option (**-O**) and location for the p4 libraries. Changing this makefile for use on, for example, IBM RS/6000's requires several changes, including additional libraries to link with (**-lbsd** in this case). The portable makefile listed above needs no changes to run on the RS/6000 or many other architectures.

1.6 Systems Supported

Chameleon supports both native (vendor) communications libraries and several popular "portable" communications packages. The "portable" packages supported are p4 [2], PICL [4], and PVM [1]; programs written with Chameleon run unchanged on any system that those "portable" packages support. Of these, p4 supports the widest variety of systems, including workstations and massively parallel computers.

There is also support for the Intel family of parallel computers, currently including the iPSC/i860, Touchstone Delta, and Paragon; IBM's SP-1 (using EUI); and the Thinking Machines CM-5, using version 2 or later of the CMMD message-passing library.

The model of parallel computation is a "hostless" MIMD (Multiple Instruction Multiple Data) one. In this model, there is no distinguished host or processor, and each processor may be running a different program. Most modern

parallel computer systems provide this model; it most closely resembles more conventional uniprocessor programming. It also discourages the use of the host to do more than start and stop the application; using the host more actively may degrade the parallel scalability of an application because the host becomes a resource bottleneck. For programs that require a host, one of the nodes can be designated the host. Processor subsets, described in Chapter 4, may be used to restrict operations to the remaining nodes.

There is also "inverse" support for some of the other communications systems. Programs using the supported subsets of PICL or Intel NX may be linked with, for example, the p4 or PVM versions of Chameleon, providing an easy way to port codes from a parallel computer to a cluster of workstations. This approach is particularly helpful for parallel computer systems with poor debugging environments; the code may be debugged in a familar environment of workstations using standard tools.

1.7 Include Files

Most C programs should use the include files '`tools.core/tools.h`' and '`tools.core/comm/comm.h`'. These should be included with

```
#include "tools.h"
#include "comm/comm.h"
```

Make sure that the C compiler is told to look for include files in the tools directory; often this is done by using the command line argument `-I/usr/local/tools.core`. (It is necessary to tell the C compiler this because the include files may include additional include files; the path names for these additional include files is relative to the root of the tools directory tree.) If you use the portable makefile system, this is done automatically.

Fortran users need not include any files; however, the file '`tools.core/comm/fcomm.h`' may be useful.

1.8 Linking

To build programs with Chameleon, you need to link with a number of libraries. For simplicity in using Chameleon for both program development and production computing, separate libraries are maintained for debugging, profiling, and production. These libraries are in the directories

debugging	'`tools.core/libs/libsg`'
profiling	'`tools.core/libs/libs0pg`'
production	'`tools.core/libs/libs0`'

To allow the libraries for many different architectures to reside on the same filesystem, the name of the architecture (such as '**sun4**' or '**rs6000**') defines an additional directory level. For example, the debugging libraries for the Sun 4 are found in the directory '**tools.core/libs/libsg/sun4**'.

There are three libraries that you may need to link with. These are '**tools.a**', '**system.a**', and '**tools<comm>.a**', where '**<comm>**' is one of '**p4**', '**pvm**', or '**picl**'. For example, a partial make file is shown below that builds the program '**example**' using p4 on a collection of Sun 4 workstations:

```
COMM = p4
BOPT = O
CFLAGS = $(BASOPT) $(COPT)
ITOOLSDIR = /usr/local/tools.core
LIBDIR    = $(ITOOLSDIR)/libs/libs$(BOPT)/sun4
example: example.o
        $(CLINKER) -o example -O example.o \
        $(LIBDIR)/tools$(COMM).a $(LIBDIR)/tools.a \
             $(LIBDIR)/system.a $(CLIB) -lm
include $(ITOOLSDIR)/bmake/$(ARCH).$(COMM)
include $(ITOOLSDIR)/bmake/$(ARCH).$(BOPT)
include $(ITOOLSDIR)/bmake/$(ARCH)
```

This builds a production version of '**example**' on a Sun 4. The **include** lines provide definitions for **CLINKER** (the linker for C programs) and **CLIB** (the communication libraries for p4 in this case), as well as the rule to compile a C program that uses the Chameleon macros (making sure the appropriate flags are defined).

1.9 Using Chameleon with Fortran

The C language is the primary language that is supported. Extensive use is made of the C preprocessor to provide much of the functionality, including traceback information that automatically includes the file name and line number. This allows the easy determination of exactly where errors have occurred during the debugging stage. Fortran is supported to a lesser extent, as there is no standardized Fortran preprocessor. However, each of the C routines listed here has a Fortran counterpart. Where pointers are used in the C version, integers should be used by the Fortran programmer (the implementation automatically handles the translation between integers and pointers).

The libraries '**tools.core/fort/$(ARCH)/fort$(COMM).a**' and '**tools.core/fort/$(ARCH)/fort.a**' provide a Fortran interface to the Chameleon routines. These libraries *must* occur ahead of the '**tools**' libraries in the link line. For example, this makefile fragment links a Fortran program (**example**) with the appropriate libraries:

```
ITOOLSDIR = /usr/local/tools.core
LDIR      = $(ITOOLSDIR)/libs/libs$(BOPT)$(PROFILE)/$(ARCH)
LIBS      = $(LDIR)/tools$(COMM).a $(LDIR)/tools.a \
            $(LDIR)/system.a -lm
FLIBS     = $(ITOOLSDIR)/fort/$(ARCH)/fort$(COMM).a \
            $(ITOOLSDIR)/fort/$(ARCH)/fort.a
include $(ITOOLSDIR)/bmake/$(ARCH).$(BOPT)$(PROFILE)
include $(ITOOLSDIR)/bmake/$(ARCH).$(COMM)
include $(ITOOLSDIR)/bmake/$(ARCH)
example: example.o
        $(FLINKER) -o example $(BASEOPTF) example.o \
                        $(FLIBS) $(LIBS) $(CLIB)
        $(RM) example.o
```

This assumes that Chameleon is installed in '`/usr/local/tools.core`'.

The interface libraries are constructed automatically from the C program files. Thus, they should always match the C versions (any new routine added to Chameleon automatically becomes available to both C and Fortran users; no special interface code needs to be written).

1.10 Chameleon and the Emerging Message-Passing Standard

The intent of Chameleon is to allow programmers to select the system that they find appropriate for the task at hand. In other words, Chameleon does *not* replace message-passing systems; it simply provides a common interface to some of the most popular systems. This common interface includes a uniform way (from the source code) to initialize a program. Thus, programs written using Chameleon are highly portable. Further, since Chameleon provides some tools for running programs written for other message-passing systems, Chameleon is a good choice for developing programs.

We hope that Chameleon can be replaced by the Message Passing Interface (MPI) standard currently under development; until that standard is specified and widely available, however, Chameleon gives programmers the widest choice of message-passing systems, as well as providing a relatively smooth migration path to the emerging MPI standard. In any event, an implementation of Chameleon that uses MPI will be provided. Further, an implementation of MPI using Chameleon will be provided (an early proposal for MPI has already been implemented in Chameleon and is described in [6]).

We note that the MPI effort does not include any standardized aids for program correctness or performance debugging. Chameleon provides these, as well as preserving any such aids provided by the underlying implementation layer (such as the trace files produced by PICL or p4).

1.11 Further Information

More detailed information about the routines mentioned in this manual may be found in the man pages (in manual section 6 "Low-level communications") or in the reference manual ('`tools.core/refs/refman.dvi`'). The script **toolman** is one of the tools provided by PETSc for accessing the detailed documentation on the routines and may be found in '`tools.core/bin/toolman`'. PETSc also provides a number of routines that may be of interest to users of Chameleon, including routines to report on floating-point errors, memory-space tracing, and debugging. See the man pages for more information.

Chameleon is continually growing through the addition of new routines. Suggestions (and bug reports) should be e-mailed to '`gropp@mcs.anl.gov`'. A users group has been set up by David Keyes; send e-mail to him ('`keyes@cs.yale.edu`') to be added to the PETSc mailing list.

Chapter 2

Starting Programs

One of the least considered but most important aspects of parallel programming models is the method by which a parallel application is started. Many approaches provide a very flexible but cumbersome approach. The approach taken here is to require as few changes to a sequential program as can be managed, whether the program is an existing application or a new design. As always, the advanced user with special needs can use vendor and/or system-specific mechanisms to provide special features (Chameleon does not prevent the user from accessing these features). The goal of the Chameleon routines is to make a parallel program look as much like a uniprocessor program as possible. For example, if a uniprocessor program, **a.out**, is run by

```
a.out <arguments>
```

then the parallel version is just

```
a.out -np <number_of_processors> <arguments>
```

This is accomplished by taking the original main program and turning it into a routine that is executed in parallel.

2.1 Getting Started

For most users, a very simple interface is suitable; just replace

```
main (argc, argv)
```

with

```
worker (argc, argv)
```

This uses a main program provided by the file '**cmain.o**' (for C programs) or '**fmain.o**' (for Fortran programs) in the tools library (for example, '**tools.core/libs/libsg/sun4/cmain.o**') that must be linked with when you link your program. A sample makefile is provided in Chapter 8. This interface provides for a wide variety of command line options; these are discussed in detail in Section 2.3. To get started, all that you need is the **-np** option which specifies the number of processors. Section 2.3.4 describes arguments that are useful in debugging programs. Fortran users should replace **program main** with **integer function worker()** and add the file '**fmain.o**' to the link step. This process is demonstrated in the sample makefiles.

2.2 PICall Interface

For more flexibility, Chameleon allows you additional control by providing the **PICall** routine. A sequential program that begins with

```
main(argc,argv)
```

is replaced by

```
main(argc,argv)
int  argc;
char **argv;
{
int worker();
/* user setup ... */
PICall( worker, argc, argv );
}
worker( argc, argv )
```

When this program is run, it will use a collection of processors. **PICall** will process some of the arguments in **argv**; these allow the user to specify the number of processors, various debugging flags, and resource limits.

2.3 Options

PICall processes a number of options in the argument list (**argc** and **argv**). These can be divided into three categories: parallelism arguments, resource limits, and debugging. The debugging options are discussed in detail in Chapter 5. Some of these are meaningful only for distributed computing (such as on a collection of workstations).

2.3.1 Parallelism Arguments

The following parallelism arguments describe the number of processes, location of the executable, and the type of machine on which to run the program.

-np n	Specifies the number of processors, **n**, to use
-arch name	Specifies the architecture to run the program on
-exes name	Specifies the full path name of the executable
-pihosts names	Specifies the names of the machines to run on. The names must be separated by commas.

Only the **-np n** argument is required; the others will be determined from the environment that the program is run on (that is, if you start a program on a Sun 4, the architecture will be **sun4** and the name of the executable will be that of the running program).

The architectures may be set with the environmental variable **TOOLSARCHES**. If, for example, you issue the shell command

```
setenv TOOLSARCHES sun4:rs6000:IRIX
```

any Chameleon program will attempt to use any machines of the type **sun4**, **rs6000**, or **IRIX** that are available.

On an MPP such as the Intel Delta, none of these arguments are needed. For example, to run the program **example** using 16 processors (on a 4 by 4 mesh) on the Delta, use

```
mexec -t "(4,4)" -f "example"
```

Here, **mexec** is the Intel NX command to start a program on the Delta. Chameleon does not specify how a program is started; rather, it strives for *source code portability*. That is, the source code is portable even if the interface to the operating system commands to start a parallel job is not.

2.3.2 Resource Limits

With any program, it is necessary to set some limits on the amount of resources used. With uniprocessor programs run on an individual's workstation, this resource limiting is often done manually by the user: if the program is taking too long or using too much memory, the user kills it. In a parallel environment, this is often difficult. When running on a collection of workstations, many (perhaps all) of the processes will be running on remote workstations. In this case, manual detection of runaway or greedy processes is impractical. To provide a simple mechanism to prevent runaway jobs, **PICall** enforces resource limits on CPU time, elapsed time, memory use, and page faults (since a code that is generating large numbers of page faults will often adversely affect the performance of a

workstation, as well as run poorly). Default limits that are suitable for small programs are provided; these limits can be overridden on the command line with the following arguments:

-cpu min	Minutes of CPU time allowed.
-mem mb	Megabytes of memory allowed.
-pf n	Number of pagefaults allowed.
-nice n	Nice increment.
-dtime hh:mm	Total elapsed time (hours:minutes) allowed.
-atime hh:mm	Absolute time by which the program must be finished (on the same day as it started).

These arguments may also be set by calling the routine `SYChangeResourceDefaults` *before* calling `PICall`.

2.3.3 Hosts File

When Chameleon is used for distributed computing, it needs to determine the processors that will be used. Several methods are provided for this. One, which will be discussed later, allows the user to list the processors to be used. However, in most cases (particularly when the distributed-computing version is being used to debug programs intended for an MPP), any available processors can be used. `PICall` uses a file, called the hosts file, to determine which computers are available and what resource limits apply to each machine. Each line of the file specifies a computer (by internet host name), architecture type, principal user, number of processors, type of processor (workstation, shared-, or distributed-memory), and resource limits. The resource limits are specified as a time period when the limits apply, and the actual limits. A particular computer may be mentioned on several lines, with each line indicating a different time period. For example, the following three lines indicate that the machine **mysun** is available for small jobs from 8 am to 6:30 pm Monday through Friday and for large jobs the rest of the time. Small jobs are defined by the third line as those using no more than 5 megabytes of memory, 8 CPU minutes, and 1000 page faults. In addition, the job will be "niced" by 9. User gropp, the principle user of the machine, may use it at any time.

```
mysun sun4 gropp 18:30-08:00 M-F  0  0    0 1 0 W
mysun sun4 gropp 00:00-23:59 S-Su 0  0    0 1 0 W
mysun sun4 gropp 08:00-18:30 M-F  5  8 1000 1 9 W
```

The next example shows a shared-memory machine with 20 processors. This machine is named **srver** and is available at all times.

```
srver symmetry root 00:00-23:59 M-Su 0 0 0 20 0 S
```

The option **-listnodes** will list the processors that have been chosen. The option **-pidbug** will indicate which processors in the hosts file were accepted and which were rejected, along with the reason for that rejection.

The location of the hosts file is given by the environment variable **TOOLSHOSTS** (for p4 programs) or **TOOLSPVMHOSTS** (for PVM programs); a default is established when the Chameleon package is installed. The entries in this file must follow some very particular rules; should you need to modify it or provide your own hosts file, try to find an example that is close to what you want. Also, be careful that you do not use other people's workstations without their permission; one of the main reasons for the hosts file database is to encourage the contribution of workstations to a pool of available computers. The principal users of a workstation are more likely to contribute their machines if they know that their workstation will not be pummeled by users of parallel programs.

2.3.4 Debugging

PICall understands a number of arguments that aid in debugging parallel programs both for correctness and for performance. These arguments are described briefly below; more details are in Chapter 5.

-trace	Enable communication tracing.
-tracefile name	Specify a file name for the tracing information to be written to; **stdout** is the default. If the name contains "%d", the value of **PImytid** (the processor number) will replace the "%d".
-event	Enable event tracing. A logfile will be written.
-eventfile name	Specify the event file. The same syntax is used as for **-tracefile**.
-summary	Enable communication summary.

2.4 Examples

This section shows how to use the command line arguments to specify different processors and informational behavior from a program.

a.out -np 4 Use four processors from any that are available

a.out -np 4 -pihosts sun2,sun3,sun4 Use the four processors consisting of the processor the program was started on and the hosts sun2, sun3, and sun4.

a.out -np 4 -cpu 1 -mem 4 Use any four processors, restricting the run to 1 CPU minute and 4 megabytes of memory (such a restriction may make more machines available).

a.out -np 4 -trace Cause all message-passing operations to write to **stdout** (see Chapter 5).

a.out -np 4 -event Produce an event file for use with Upshot [8].

a.out -np 4 -event -blogfmt picl Produce an event file for use with ParaGraph [7].

2.5 In Case of Trouble

On a collection of workstations, the most common cause of trouble is having too few workstations available. **PICall** uses a database of workstations (described in Section 2.3.3) to guide the selection of available machines. This database includes resource limits on CPU, memory, and pagefaults. Different limits may be established for different times of day. Through the use of this database, the workstation owners (the principal users of the workstations) are assured that their machines won't be swamped with remote jobs when they need them; conversely, their machines can be made available for short development tests during the day and for long production runs at night.

The most common cause of the message

```
Could not find enough acceptable processors
```

is that there are not enough processors offering the requested resources. Try specifying smaller values of **-cpu** and **-mem** on the command line. If this does not work, make sure that the database contains enough processors of the requested architecture. The default database file is '**tools.core/comm/hosts**'. The file name can be overridden by specifying the environment variable **TOOLSHOSTS** for p4 and **TOOLSPVMHOSTS** for PVM.

If the parallel job seems to start but then hangs, there may be a problem with the workstations or your program's access to them. Unless you are using the p4 server, p4 requires that **rsh** work; try doing **rsh <workstation> ls** for each workstation that you wish to use. When p4 is used, a temporary file is created with a name that begins with the characters **PI**, for example, **PIa1037**. This file is a p4 procgroup file and lists the machines being used. On successful completion of a run, this file is removed. If the program hangs, you may refer to this file to help determine which machine may be causing the problem.

PVM 2.4.x has a particular feature that can sometimes cause a job to hang. If a program aborts or is killed in a way that does not cause all of the participating processes to execute the PVM **leave** routine, it is impossible to rerun that

program successfully without restarting the PVM daemon. This problem is caused by the way parallel programs in PVM get access to each other through the **enroll** routine. If you suspect that this is the problem, kill and then restart the PVM daemon and then try running your code.

Finally, both p4 and PVM benefit from starting a "server" before running any parallel programs. Sample shell scripts that use the host database to start the servers is in '**tools.core/comm/daemons**'. These may be run by any user. It is not necessary to run these scripts, but doing so may significantly reduce the time that it takes to start a parallel application.

If all of this fails, look at Chapter 5 on debugging. The options **-trace** may be used to indicate where a program is getting hung.

Chapter 3

Message Passing

Message passing is a well-known and portable method for writing parallel programs. This document does not describe the technique; rather, it describes a particular portable implementation. This implementation is designed to make the full power of a message-passing system available; it is not a least-common-denominator design.

As an aside, if you are writing an application program, you may not need any of these routines (except perhaps the inquiry routines). Instead, you should see whether any higher-level communication or computation routines (such as a parallel linear solver) meet your needs. If you do not find the routines you need, let us know. They may be available elsewhere, or they may be general enough that they could be added to Chameleon.

On systems that do not support certain message-passing features, such as nonblocking communications, Chameleon will automatically emulate the behavior (as much as possible). Thus, one need not give up efficiency on a particular machine in order to obtain portability. Chameleon also provides a way to determine which features are supported, in the event that different algorithms would be used depending on the available features.

3.1 Getting Started

It is not necessary to master all of these routines to write a working parallel program. To begin with, use the blocking message-passing routines with user-specified buffers. These routines are

```
PIbsend(tag,buffer,length,to,datatype);
PIbrecv(tag,buffer,maxlength,datatype);
```

The first line sends a message, and the second line receives one. The parameters are

tag	User-specified message tag (often called the message *type*). This should be a non-negative integer; it should also not be too large (the routine **PITagRange** will give the range of allowable tags).
buffer	Pointer to the buffer to send (for **PIbsend**) or to receive into (**PIbrecv**).
length	Length of the message to send, in bytes.
maxlength	Maximum length of a message to receive, in bytes. The routine **PIsize** may be used to determine the actual size.
to	Processor id of the processor to send to. This is an integer between 0 and **PInumtids**-1. The id of a processor is given by **PImytid**.
datatype	Datatype of the message. The use of this parameter to build programs that are portable to heterogeneous collections of processors is discussed below. A table of possible values is given below.

3.2 Overview

In Chameleon, message passing means sending a buffer of data (the message) with a user-defined tag from one processor and receiving it on another processor. The choices of buffer, the kind of sending semantics, and the time when the buffer becomes available for reuse are made by selecting one of a set of message-passing routines (all with nearly identical calling sequences). This section describes the routines for sending messages consisting of contiguous bytes. This is the most common type of message.

The names of the message-passing routines (actually macros in C) have the following general format:

```
PI<blocking?><operation><user_buffer?><fast_protocol?>
```

Below we describe the choices for each of the four fields.

The operations are

send	Send a message.
recv	Receive a message.

The values for **blocking?** are

b Blocking operation. When the routine completes, the operation has been performed. For a send, this means that the message has been sent (but not necessarily received). For a receive, this means that a message has been received.

n Nonblocking operation. When the routine completes, a handle has been set. It is necessary to use the **wait** modifier with this handle to ensure that the operation (both sends and receives) has completed.

w For nonblocking operations, wait until a previously requested operation completes.

The values for **user_buffer?** are

<null> Indicates that the buffer was not allocated with the message buffer allocation routines (see **PINewSendBuf** and **PINewRecvBuf**). This is the usual case.

m Indicates that the buffer was allocated with the buffer allocation routines.

Here, **<null>** means "blank".

The values for **fast_protocol?** are

rr Indicates that a fast but possibly unreliable message protocol is to be used. A message sent using this modifier may be discarded if the destination processor has not already executed an appropriate receive, or an application may block until the destination executes a receive for this message. The "rr" stands for "ready receiver."

<null> Indicates that a correct but possibly slower protocol should be used for messages.

In addition to these routines, there are routines to determine whether messages are available and whether a nonblocking routine has finished. These are also described below.

All of the routines that send or receive data take a **datatype** argument. This indicates what kind of data the message contains; it is used to allow the use of heterogeneous collections of machines which may use different storage formats for integers, floating-point values, etc. The valid datatypes are as follows:

Datatype	C	Fortran
MSG_SHRT	short	
MSG_INT	int	integer
MSG_LNG	long	
MSG_FLG	float	real
MSG_DBL	double	double precision
MSG_OTHER	char	character

MSG_OTHER should be used for any data whose type is unspecified.

This raises the issue of what to do if a message containing different datatypes is to be sent, for example, a structure defined as

```
struct {
    int n, m;
    double a, b;
    }
```

If you are using machines that all use the same storage formats, and you do not want portability to collections of machines with different formats, use **MSG_OTHER**. (You should mark this in your code; such assumptions can cause difficult maintenance problems.)

3.3 Blocking Message-Passing

Blocking message-passing is the simplest form of message passing. The term blocking here refers to the message buffer: when the routine exits, the buffer is ready for use. In the case of a send, this means that the buffer may be reused. For a receive, this means that the buffer contains the received data. The basic routines are

PIbrecv — Receive a message of a given tag into a buffer that was allocated by the user.

PIbsend — Send a message of a given size (in bytes) to another processor.

PIbrecvm — Receive a message of a given tag into a buffer (allocated with **PINewRecvBuf**)

PIbsendm — Send a message of a given size (in bytes) to another processor. The buffer must have been allocated with **PINewSendBuf**.

PIbrecvUnsz — Receive a message of unspecified length. **PIbrecvUnsz** allocates a buffer for the message and returns a pointer to the allocated buffer and the size of the message.

The formats are

```
PIbrecvm(tag,buffer,maxlength,datatype)
PIbsendm(tag,buffer,length,to,datatype)
PIbrecvUnsz(tag,&buffer,&size,datatype)
```

The parameters are

tag	Message tag
buffer	Pointer to buffer
datatype	Datatype (e.g., `MSG_INT`)
length	Size of message to send in bytes
maxlength	Size of the buffer in a receive (allows a message of this size or smaller) in bytes
size	Size of a received message in `PIbrecvUnsz` in bytes

For more details, see the man pages or the reference manual.

The variants such as `PIbsendrr` have the same calling sequences but the slightly different semantics, as described in the overview.

There are four such routines:

PIbrecvrr	PIbrecvmrr
PIbsendrr	PIbsendmrr

3.4 Nonblocking Message-Passing

Nonblocking message-passing allows the programmer to overlap communication and computation (when the underlying hardware and system software supports it). This is done, in the case of a send, by specifying a buffer to send. The state of the buffer becomes undefined and may not be changed by the programmer until the send operation completes.

A sample use of a nonblocking send is

```
PISendId_t id;
...
PInsend(tag,buffer,length,to,datatype,id)
... much computation ...
PIwsend(tag,buffer,length,to,datatype,id)
```

The programmer may not use the buffer until the `PIwsend` completes (the contents at that time are implementation defined). The parameter `id` (which is not present in the blocking version `PIbsend`) is an identification value that permits multiple nonblocking sends to be issued before their corresponding `PIwsends`.

In the case of a nonblocking receive, the programmer tells the system where to put a message when it arrives. This can save memory references and can significantly improve the performance of a parallel program. A sample use is

```
PIRecvId_t id;
...
PInrecv(tag,buffer,maxlength,datatype,id)
... much computation ...
PIwrecv(tag,buffer,maxlength,datatype,id)
```

The "wait" has the same parameter list as the "nonblocking" operation in order to make it easy to simulate nonblocking operations with blocking ones. It also aids the programmer in seeing just what data was expected at the end of a nonblocking operation; this allows easy runtime checking that the expected operation was performed.

The variants such as **PInrecvm** and **PInsendrr** have the same calling sequences but the slightly different semantics, as described in the overview.

There are twelve such routines:

PInrecvm	PInrecvrr	PInrecvmrr
PInsendm	PInsendrr	PInsendmrr
PIwrecvm	PIwrecvrr	PIwrecvmrr
PIwsendm	PIwsendrr	PIwsendmrr

In addition to these routines, the routine **PInstatus** may be used to determine whether a particular nonblocking message operation has finished. The format is **PInstatus(id)**, where **id** is the identification value from the **PInsend** or **PInrecv**. **PInstatus** returns the value 1 if the message is completed and 0 otherwise.

Note: if nonblocking messages are not supported by the underlying system software, the use of the pairs **PInsend()...PIwsend()** and **PInrecv()...PIwrecv()** will still work. However, **PInstatus** in those cases will always return 0.

The macro **PI_NO_NSEND** is defined if there is no nonblocking send; **PI_NO_NRECV** is defined if there is no nonblocking receive. These may be used to select different algorithms in the two cases at compile time.

3.5 Information about a Message

When a message is received, the basic routines simply return. Sometimes, additional information is required, such as who sent the message, how long it is, and what message tag it had. These are available from the following routines:

PIfrom	Return the processor id of the sender of the message.
PIsize	Return the length of the message in bytes.
PItag	Return the tag of the message.

These all return information about the last message received; they should be called at most once for each message. Each is a function that takes no parameters (i.e., use `PIfrom()` to determine which processor sent the last message that was received).

This approach is not "thread safe." That is, it can cause problems if there are multiple threads of control. This situation can happen if, for example, a signal or interrupt handler issues a receive. The MPI draft standard provides a thread-safe version; the next version of Chameleon may provide a thread-safe version of the receive routines (in an upward compatible way, of course).

Two routines allow the programmer to determine whether a message of a given tag is available. The routine `PInprobe` returns 1 if so, and 0 if not. The routine `PIbprobe` does not return until a message of the specified tag becomes available. The formats of these routines are as follows:

`PInprobe(tag)`	Nonblocking test for a message of the given tag
`PIbprobe(tag)`	Blocking test for a message of the given tag

To receive a message that has been probed for, use `PIbrecvProbed` rather than `PIbrecvm`. The former is needed because of race conditions in some implementations of `PIbprobe` and `PInprobe`. The format of `PIbrecvProbed` is

```
PIbrecvProbed( tag, buf, maxlength, datatype )
```

3.6 Message Buffers

For the best performance on some systems, buffers that are used for sends and receives should be allocated with the routines `PINewSendBuf` and `PINewRecvBuf`. These routines allow the underlying implementation to use special buffer formats or structures for improved performance; if these routines are not used to allocate buffers, be sure to use the `user_buffer=<null>` versions of the send and receive routines. (Fortran users cannot allocate memory this way, so they cannot use the `m` versions.)

`PINewSendBuf`	Allocate a send buffer. The format is `PINewSendBuf(buf,size,type)`, where `type` is a valid C type. For example, to create a buffer `buf` for 32 `doubles`, use `PINewSendBuf(buf,32*sizeof(double),double)`.
`PINewRecvBuf`	Allocate a receive buffer. The format is the same as `PINewSendBuf`.
`PIFreeSendBuf`	Free a buffer allocated with `PINewSendBuf`. The format for freeing a buffer `buf` is `PIFreeSendBuf(buf)`.
`PIFreeRecvBuf`	Free a buffer allocated with `PINewRecvBuf`.

3.7 Machine Topology

Some parallel programs need to know something about the number of processors and the interconnection of processors on the parallel machine on which they are running. Other information, such as the ranges of valid message sizes and tags, is needed by general-purpose programs. Virtually all programs need to know the id of the running process. This information is provided by the following routines:

`PInumtids`	Returns the number of processors
`PImytid`	Returns the index of the processor, in the range 0 to `PInumtids`−1
`PIdistance`	Gives the distance between two processors. The format is `PIdistance(from,to)`. The returned value is the number of hops from processor `from` to processor `to`. The exact definition of a hop is implementation dependent; the only requirement is that `PIdistance(from,from)` is zero. Note that an implementation may define `PIdistance(from,to)` to be zero even for `from` different from `to`. The value must be non-negative.
`PIdiameter`	Returns the maximum value of `PIdistance` for all pairs of processors. This is the "diameter" of the parallel processor.
`PIMsgSizes`	Returns the minimum and maximum message sizes. The format is `PIMsgSizes(&min,&max)`; the values `min` and `max` are in bytes.
`PITagRange`	Returns the range of valid message tags. The format is `PITagRange(&low,&high)`. Note that some systems have a very limited range of tags.

Most of these routines are actually macros that return simple values.

3.8 System Functionality

C programmers can access, through macros, information on the capabilities of the parallel system at compile time. These macros are

PI_NO_NSEND Defined if nonblocking send is not supported by the transport layer

PI_NO_NRECV Defined if nonblocking receive is not supported by the transport layer

PI_NO_READYRECEIVE Defined if ready-receive is not supported by the transport layer

PI_NO_NATIVE_GLOBAL Defined if the transport layer does not directly support collective operations on all processors

In all cases, Chameleon will simulate the capability if the underlying transport system does not provide it.

Chapter 4

Collective Operations

Many programs require operations that involve a collection of processes. When every processor is involved, these are called global operations. Examples include barriers (all processors wait until all have reached the barrier), sums (all processors contribute to a sum), and broadcasts (one processor sends a value to all others). These operations also make sense on any subset of processors. The routines described here allow the programmer to dynamically define subsets of processors. All of the collective operations are defined on these subsets. For simplicity, these routines are described first for collective operations on all processors by using the predefined processor set **PSAllProcs** that contains all of the processors. Section 4.6 describes how to define subsets of processors. Section 4.10 discusses where and why processor sets are useful.

4.1 Getting Started

The most common use of collective communication routines is on all of the processors (as opposed to a subset of the processors). The routine

```
PIbcast( buf, size, issrc, PSAllProcs, datatype )
```

broadcasts a buffer **buf** of **size** bytes to all processors; the value of **issrc** is **1** for the sender and **0** for all other nodes.

The routine

```
PIgdsum( val, n, work, PSAllProcs )
```

sums **val** (an array of size **n** of doubles) across all processors, returning the sum in **val**. The array **work** of size **n** doubles must be provided.

There are additional routines for finding the maximum, minimum, and logical operations on various datatypes.

4.2 Reductions

A common operation in parallel programs is the reduction of a value (or collection of values) held on many processors into a single value (or collection of values). For example, each processor may compute a value `v` and desire the minimum of all the `v`'s on all processors. If `v` is a double, the routine `PIgdmin` will compute that minimum. Since a reduction requires combining values, it is necessary to know what datatype and format is being combined. These datatypes are double, int, float, etc. To simplify the routines, the second character in the routine name is used to denote the datatype.

These characters are

d	double
i	int
f	float
c	char

The routines are

`PIgXsum`	Finds the sum of the values across the processor set.
`PIgXmax`	Finds the maximum of the values across the processor set.
`PIgXmin`	Finds the minimum of the values across the processor set.
`PIgXor`	Finds the logical "or" of the values across the processor set (`X` may be `i` or `c` only).
`PIgXand`	Finds the logical "and" of the values across the processor set (`X` may be `i` or `c` only).

The last two routines are not defined for double or float types.

The format of all of these routines is the same: `PIgxxxx(val,n,work,procset)`, where `work` is a work area of the same size as `val`, and `n` is the number of elements. For example, to find the maximum value of a number on all processors, use

```
double val, work;
...
PIgdmax( &val, 1, &work, PSAllProcs );
```

When any of these routines exits, `val` contains the result.

Some systems separate the meaning of reduction from combination; in one case, a single node gets the reduced value; in the other, all nodes get the value. Chameleon currently provides only the version where the final value is made available to all nodes.

The routine `PIcombine` will combine values using a user-provided routine. The format of this routine is

```
void PIcombine( val, n, work, procset, elmsize, datatype, op )
void    *val, *work;
int     n, elmsize, datatype;
ProcSet *procset;
void    (*op)( );
```

where the routine op is defined as

```
void op( a, b, n )
void *a, *b;
int  n;
```

and the result of op is to perform the operation $a \leftarrow a$ op b. In this way users may easily write their own global operations, for instance, multiplication, which will be as efficient as the standard global operations.

4.3 Broadcast

Broadcast routines send a buffer to all other nodes in the designated processor set. The source of the buffer can be indicated in two ways. With **PIbcast**, exactly one processor should set the argument **issrc** to 1; all others use 0. This is appropriate for programs where the source of the buffer is not known. The format of this routine is

```
PIbcast( buf, size, issrc, procset, datatype )
void    *buf;
int     size, issrc, datatype;
ProcSet *procset;
```

If the source is known, the routine **PIbcastSrc** should be used. This routine takes an argument **src** that is the processor id (from **PImytid**) of the processor that is the source of the data to scatter.

The format of this routine is

```
PIbcastSrc( buf, size, src, procset, datatype )
void    *buf;
int     size, src, datatype;
ProcSet *procset;
```

Care should be taken in using these routines. All processors in the selected processor set must call the same routine (either **PIbcast** or **PIbcastSrc**). If there are multiple, nondisjoint processor sets, all of which are using a **PIbcast**, it is necessary to ensure that there is no possibility of deadlock. For example, if there are two processor sets **set1** and **set2**, each containing the 2 processors 0 and 1, then the code

```
if (PImytid == 0)
    PIbcast( ..., set1, ... );
    PIbcast( ..., set2, ... );

else
    PIbcast( ..., set2, ... );
    PIbcast( ..., set1, ... );
```

will deadlock.

4.4 Gather

Gather routines allow data from all nodes to be gathered to all nodes. Each processor contributes some local data (in **lbuf**) and, at the completion of the routine, receives the collected data (in **gbuf**). The routine **PIgcol** handles the case of data of variable and unknown size; **PIgcolx** handles the case of data of known size.

The format of **PIgcol** is

```
PIgcol( lbuf, lsize, gbuf, gsize, glen, procset, datatype )
void    *lbuf, *gbuf;
int     lsize, gsize, *glen, datatype;
ProcSet *procset;
```

The parameters are

lbuf	Buffer to hold the local contribution
lsize	Number of bytes in the lbuf
gbuf	Buffer to hold the result
gsize	Size of gbuf in bytes
glen	Actual number of bytes received
procset	Processor set to collect over
datatype	Datatype of lbuf and gbuf

The order of the collected data is implementation defined.

The format of **PIgcolx** is

```
PIgcolx( lbuf, gsizes, gbuf, procset, datatype )
void    *lbuf, *gbuf;
int     *gsizes, datatype;
ProcSet *procset;
```

The parameters are

lbuf	Buffer to hold the local contribution
gbuf	Buffer to hold the result
gsizes	Array of the sizes of each contribution (`gsizes[PImytid]` The size of the local contribution)
procset	Processor set to collect over
datatype	Datatype of lbuf and gbuf

The data is collected in order of node number. For processor subsets, the order is that returned by `PSPROCLIST`.

4.5 Barriers

Often, it is necessary to have all processors stop until they all have reached a common point in a computation. Such an operation is called a barrier, rendezvous, or synchronization. This is provided by the routine `PIgsync`. The format is `PIgsync(procset)`.

As a special case of a barrier, it is sometimes necessary to allow only one processor to access a resource at a time. For example, if each processor is to write to a file (or to standard output), it is necessary to ensure that only one writes at a time. A simple way to do this is with a barrier:

```
for (i=0; i<PInumtids; i++) {
    PIgsync(PSAllProcs);
    if (i == PImytid) {
        <myoutput>
        }
    }
```

However, this can be very inefficient. The routine `PIgtoken` provides an alternative approach. The same effect as above is achieved with

```
for (i=0; i<=PInumtids; i++) {
    if (PIgtoken(PSAllProcs,i)) {
        <myoutput>
        }
    }
```

The routine `PIgtoken` passes a "token" from one processor to the next in sequence, returning the value 1 when a processor receives the token. The actual value of the token may be set with `PISetToken` and retrieved with `PIGetToken`.

4.6 Processor Sets

A processor set is a collection of processors. Usually, this is a subset of the total number of processors available. To the user, a processor subset is just a pointer to a structure; the internal contents of this structure are used by the library to perform the requested operations. When the processor subset is no longer needed, the subset should be deleted. As a practical matter, using the three-phase process of creating a subset, using the subset, and deleting the subset allows the library to spend extra effort to optimize the use of the subset. This optimization may include choosing special communications schedules for the collective operations.

A processor set is created with the routine **PSCreate**. Each processor set must have a unique name that all processors in the set agree on.

To use these routines, create a processor subset (**procset**) and specifiy which processors are in that subset. This **ProcSet** is then passed to the collective operation routines. A predefined set, containing all of the processes, is **PSAllProcs**. An example of this process is

```
ProcSet *procset;
int     name;
...
name    = 5;
procset = PSCreate( name );
```

When a processor set is created, it has no members. To add members, use **PSAddMember**. This takes an array of processor numbers and a processor subset and adds those processors to the set. It may be called multiple times to add groups of processors. The following two examples add to a processor set the even-numbered processors:

```
int i;
...
for (i=0; i<PInumtids; i += 2)
    PSAddMember( procset, &i, 1 );
```

and

```
int i, *p;
...
p = (int *)MALLOC( ((PInumtids + 1) / 2) * sizeof(int) );
for (i=0; 2*i<PInumtids; i ++)
    p[i] = 2*i;
PSAddMember( procset, p, i );
FREE(p);
```

Before a processor set is used in a collective operation, it must be "compiled." That is, once the members of the processor set are defined, the various internal

fields that are used by the collective operations must be calculated. The routine **PSCompile** is used for this. Once **PSCompile** is called, no other processors may be added to the processor set. The following example creates, defines, and compiles a processor set consisting of the even-numbered processors:

```
ProcSet *procset;
int     i, name;
...
name    = 5;
procset = PSCreate( name );
for (i=0; i<PInumtids; i += 2)
    PSAddMember( procset, &i, 1 );
PSCompile( procset );
```

When the processor set is no longer needed, use **PSDestroy** to remove it: `PSDestroy( procset );`.

While the use of **PSCreate**, **PSAddMember**, and **PSCompile** is completely general, it requires that all processors in the processor set know the processor ids of all of the processors that are to be in the set. Sometimes we wish to form a set by collecting all of the processors that want to be in the set, without each processor knowing in advance the members of the set. This is done with **PSPartition**. This routine takes a value (a name for the processor subset) and finds all of the processors with that name. All processors are put into at most one processor set by this routine. For example, the following code creates a processor set consisting of the even-numbered processors (on even-numbered processors) and of the odd-numbered processors (on odd-numbered processors). That is, if there are eight processors, this will create two disjoint processor sets: **0 2 4 6** and **1 3 5 7**, with the even-numbered processors creating the first of these and the odd-numbered creating the second of these processor sets.

```
ProcSet *procset;
procset = PSPartition( PImytid % 2, PSAllProcs );
```

A negative number for **value** excludes that task from any of the new processor sets.

Note that **PSPartition** takes a processor set as the second argument. This allows the user to partition a subset of processors. The value **PSAllProcs** is the processor set containing all of the nodes in the parallel machine.

The routine **PSUnion** may be used to combine two processor sets to form a new processor set.

For many applications, it is necessary to know something about the members of a processor subset, such as who they are, how many processors are in the set, and who their neighbors are. The routines described in this section gives access to the information in a processor subset.

PSnumtids The routine **PSnumtids(procset)** returns the number of processors in the processor set.

PSmytid	The routine `PSmytid(procset)` returns the *relative* id or rank of the processor in the processor set. This value is between 0 and `PSnumtids(procset)`-1.
PStidFromRank	The routine `PStidFromRank` returns the process id corresponding to a given rank in a processor set.
PSPROCLIST	The routine `PSPROCLIST(procset,list)` puts the processor ids of the processors in the procset into the array list (`int list[]`).
PSISROOT	The routine `PSISROOT(procset)` returns 1 if the processor is the "root" of the processor set. For the processor set of all nodes, this is equivalent to the expression `PImytid==0`.
PSROOT	The routine `PSROOT(procset)` returns the processor id of the root of the processor set. `PSROOT(PSAllProcs)` is equivalent to `0`.
PSSetMeshSize	The routine `PSSetMeshSize(procset,nx,ny)` specifies the dimensions of the processor set when considered as a two-dimensional mesh.
PSMESHLOC	The routine `PSMESHLOC(procset,i,j)` sets `i` and `j` to the relative location of the processor in a two-dimensional mesh (as defined by `PSSetMeshSize`).
PSNbrRing	The routine `PSNbrRing` returns the processor ids of neighbors in the processor set when considered as a one-dimensional ring. See the man page for more details.
PSNbrMesh	The routine `PSNbrMesh` returns the processor ids of neighbors in the processor set when considered as a two-dimensional mesh. See the man page for more details.
PSNbrTree	The routine `PSNbrTree` returns the processor ids of neighbors in the processor set when considered as a binary tree. See the man page for more details.

4.7 Changing the Virtual Topology

The routines used to compute the neighbors (`PSNbrRing`, `PSNbrMesh`, and `PSNbrTree`) may be changed by the user. These routines will be documented in

the next release of this documentation. The routine **PISetNbrRoutines** may be used to do this. Note that since the neighbors returned by these routines are used by the collective communications routines (such as **PIgdsum** and **PIgbcast**), changing these routines has the side effect of changing the communication patterns used by the default collective communication routines.

4.8 Changing the Reduction Method

The method used to perform a reduction may be changed by the user. Several methods are available, and advanced users may write their own and insert them into Chameleon (without changing the libraries). The routine **PISetCombFunc** sets the routine used for the reduction operations. The format of this routine to pass to **PISetCombFunc** is

```
void MyGsetop( val, n, wrk, procset, elmsize, datatype, op )
void    *val, *wrk, (*op)();
int     n, elmsize, datatype;
ProcSet *procset;
```

where op has the form

```
void op( val, work, n )
void *val, *work;
int  n;
```

The arguments are

val	Value to contribute to reduction on input and final value on output
n	Number of elements in **val**
work	Work area of the same size as **val**
procset	Processor set to work in
elmsize	Number of bytes per element of **val**
datatype	Datatype of **val**
op	Routine to combine values together

The format of **PISetCombFunc** is

```
void PISetCombFunc( func )
void (*func)();
```

4.9 Changing the Broadcast Method

The method used to perform a scatter may be changed by the user. Several methods are available, and advanced users may write their own and insert them into Chameleon (without changing the libraries). The routine `PISetScatterFunc` sets the routine used for the scatter operations. The format of the routine to pass to `PISetScatterFunc` is

```
void gscatterset( buf, size, issrc, procset, datatype )
char    *buf;
int     size, issrc;
ProcSet *procset;
int     datatype;
```

The arguments are

buf	Data to scatter (if `issrc=1` or buffer to hold data (if `issrc=0`)
size	Number of bytes in `buf`
issrc	One if the processor is the source, zero otherwise
procset	Processor set to work in
datatype	Datatype of `val`

The second argument to `PISetScatterFunc` is the routine to be used for `PIgscattersrc`; this routine has the same format, but with `issrc` replaced with `src`, the node number of the source.

The format of `PISetScatterFunc` is

```
void PISetScatterFunc( func, funcsrc )
void (*func)(), (*funcsrc)();
```

4.10 Why and How to Use Processor Sets

Many parallel computations, particularly on relatively small amounts of data, cannot effectively use large numbers of processors. In this case, it is useful to define a subset of processors for best efficiency. In other cases, a truly MIMD algorithm may want to use a cluster of processors for each of several tasks; again, the program needs to be able to perform collective operations on subsets of processors.

An example is shown in Figure 4.1. This shows that the optimal number of processors for the solution of a banded linear system by direct elimination can occur at very small number of processors (about 4 for these choices of the parameters). Using more than 16 processors takes more time than using a single

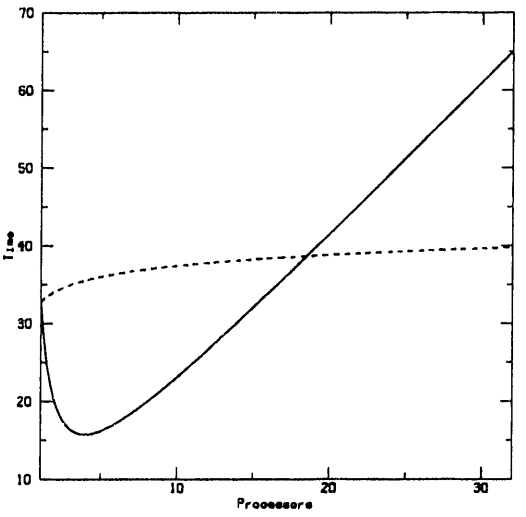

Figure 4.1: Scalability graph for banded linear system solve

processor in this case. If this banded linear system algorithm is implemented with processor sets, the optimal number of processors can always be used.

All collective routines take **procset** as an argument. By always using a **procset** argument (instead of **PSAllProcs**) and by using the processor set versions of **PImytid** and **PInumtids**, you can write code that will run on any subset of processors.

Instead of...	use
PImytid	**PSmytid(procset)**
PInumnodes	**PSnumnodes(procset)**
PImytid==0	**PSISROOT(procset)**

To convert from rank to id, use **PStidFromRank(procset,rank)**.

Chapter 5

Debugging

Debugging a parallel program can be difficult. Many of the most powerful debugging aids for uniprocessors (such as **dbx**) either are unavailable on parallel computers or have serious limitations. Further, parallel programs offer new ways to write both incorrect and poorly performing programs. This chapter discusses some of the aids provided by Chameleon to aid in debugging a program, both for correctness and for performance.

5.1 Getting Started

The debugging aids provided by Chameleon are available through the command line arguments that are processed by **PICall** or the **worker** interface. The arguments that are most commonly useful are as follows:

-trace Cause all send and receive routines, as well as many collective communication routines, to write a message to standard output indicating the size, destination, and tag of a message.

-summary Produce a summary listing when the program finishes with the amount of time spent communicating and the amount of data sent.

-event Generate an event log that can be used as input to the **upshot** [8] program analysis tool.

Additional arguments may be used to modify the behavior of these arguments (for example, **-tracefile** redirects the trace information to the specified file or files).

5.2 Correctness

Two main results of errors in message-passing parallel programs are not present in uniprocessor programs. These are deadlock (program "hangs") and nondeterministic behavior (different results for the same input data in different runs). Both of these are often caused by the program's receiving a message intended for either another processor or another part of the program. These problems are difficult to diagnose, particularly since the behavior may be difficult to reproduce. The command line argument **-trace** can help. When this argument is used, every time a message is sent or receive, a description of the operation is written to standard output. This output includes the name of the operation (**send**, **recv**), the tag of the message, and the file name and line number in that file where the operation occurred. The option **-tracefile name** may redirect this output to a file or to a different file for each processor. These logs of the communication behavior may then be examined to see whether there is a problem. In the case of deadlock, often the best way to proceed is to start at the ends of the files and work backwards. In the case of nondeterministic behavior, collect logs for several runs on the same input data, and then use **diff** to find the first place that the logs differ.

Some sample tracing output is displayed below. This shows two processors both waiting for a receive:

```
[0] recvstart <Tag 1> [buggy.c:17]
[1] recvstart <Tag 1> [buggy.c:17]
```

(This is from the ring example below, but with all processors executing their **PIbrecv** before the **PIbsend**.) The two lines from file '**buggy.c**' at line 17 are the beginning of the **PIbrecv**s. Since no processor is sending, the program hangs at this point.

Most collective operations are indicated by a **Starting <name>** and **Ending <name>**. The format of tracefile lines is

```
[processor-id] operation <data on operation> [filename:line]
```

For example, a send operation from processor 13 to processor 19, with a buffer of size 120, message tag 27, from line 193 in file myprog.c would generate the line

```
[13] send <Tag 27(120) To 19> [myprog.c:193]
```

A receive will generate two entries—one when the receive starts and one when it completes. A receive on processor 19 with message tag 27 and at line 273 in file myprog.c will generate the lines

```
[19] recvstart <Tag 27> [myprog.c:273]
[19] recv <Tag 27> [myprog.c:273]
```

If too much data is being generated to conveniently look at, consider using the `-tracefile filename` option. For example, the command line options `-trace -tracefile log.%d` will generate a separate log file for each processor. The cshell script

```
#! /bin/csh
foreach file (log.*)
    echo "$file"
    tail -1 $file
end
```

will print out the last operation that each processor was attempting. Note that using a single filename for the output may cause output to be lost because of the way many operating systems deal (or fail to deal) with multiple processes writing to the same file.

5.3 Error Messages

The debugging version of the Chameleon package will generate error tracebacks of the form

```
Line linenumber in filename: message
Line linenumber in filename: message
...
Line linenumber in filename: message
```

The first line indicates the file where the error was detected; the subsequent lines give a traceback of the routines that were calling the routine that detected the error. A message may or may not be present; if present, it gives more details about the cause of the error.

The production libraries are often built without the ability to generate these tracebacks (or even detect many errors). So, if a program crashes without warning, try recompiling with the `BOPT=g` option and then rerunning it.

5.4 Performance

Performance debugging can be very difficult. A parallel program has many sources of inefficiency, including

- poor algorithm,
- poor implementation of the computational part of the algorithm,
- poor implementation of the communication part of the algorithm,
- insufficient overlap of communication and computation,

- poor load balance, and
- malfunctioning hardware.

A number of tools are provided to help diagnose these problems. Unfortunately, the most important cause of poor performance, a poor algorithm, is the hardest to diagnose. In fact, a poor algorithm may appear to perform well by doing large amounts of highly parallel but unnecessary work, thus achieving high parallel efficiency.

Before spending a lot of time tracking down a performance problem, you should have some feel for how well you expect your program to behave. Often, this can be done by constructing a simple complexity model of the computation. Into this model will go three machine-dependent parameters: the time to compute an operation (for floating-point programs, the time for a floating-point operation), the latency in a message between two processors (the time to send a message of length zero), and the incremental time to send a byte between two processors. This is a crude model, but it often gives a good estimate of the performance of a computation. Some examples of this approach are shown in [5, 3]. To get these machine-dependent parameters, the program **twin** in '**comm/examples/angst**' can be run. The argument **-help** to **twin** gives information on using **twin**.

You should also check that there is no hardware problem on your parallel computer. If each processor does not compute at the same rate as all other processors, or if the time to send a message between two (adjacent) processors is not independent of the processors, then a correct program will appear to have a performance problem. Chameleon provides a program to test for this case (which has been observed on a number of parallel computers). Note that this problem is particularly insidious, as the program may work correctly, just not as efficiently as expected. The program is **tcomm** and is in the directory '**tools.core/comm/examples/angst**'. The argument **-help** to **tcomm** gives information on using **tcomm**.

5.4.1 Computation Implementation

A good way to determine the quality of the implementation is to compute the computational rate (megaflops) achieved by the code and to compare that with standard relevant benchmarks. For example, a program that uses primarily floating-point operations can be compared to the LINPACK benchmarks. (Note that some well-designed programs can greatly exceed the computational rates indicated in the LINPACK benchmarks.) The easiest way to compute this value is to compute (or estimate) the number of operations and divide by the time taken by those operations. However, care should be taken to ensure that meaningful times are collected. For pure floating-point operations, use **SYGetCPUTime**; this measures the CPU time taken by a user process. For example,

```
double precision SYGetCPUTime, t1, t2
...
t1 = SYGetCPUTime()
<computation to measure>
t2 = SYGetCPUTime() - t1
print *, 'Took ', t2, ' seconds'
```

When measuring communication, it is important to remember that the time spent waiting to receive a message will usually not be charged to a process and thus will not show up in the CPU time. In this case, use **SYGetElapsedTime** instead of **SYGetCPUTime**.

An alternative to **SYGetElapsedTime** that can provide higher-resolution, lower-overhead timings involves the routines **SYusc_clock** and **SYuscDiff**, for example,

```
double precision SYuscDiff, t1(2), t2(2), t
...
call SYusc_clock( t1 )
<communication to measure>
call SYusc_clock( t2 )
t = SYuscDiff( t1, t2 )
print *, 'Took ', t, ' seconds'
```

5.4.2 Communication Implementation

There are several potential sources of performance problems in the communication implementation. These include

- long transit times,
- long startups, and
- messages arriving out of order.

To diagnose these effects, use the **-event** and **-summary** switches. The **-summary** switch generates a summary listing for all processors, indicating how much time was spent sending and receiving data and what the average transfer rate was. The time spent in collective operations is also displayed. For example, here is the summary for node 2 after a run.

```
Summary for node 2:
Op:          calls      bytes    total time Average rate
Send:         3366    1689600    3.1601e+00   5.3466e+05
Recv:         3367    1689624    3.5450e+01   4.7662e+04
Global:         67          0    1.0890e+01   0.0000e+00
```

Look for

- low transfer rates but long messages, and
- large times (relative to the computation time).

Low transfer rates for long messages may indicate that much of the time was spent waiting for messages to arrive (including waiting before they were sent). This may indicate either a load imbalance or contention in the communication network. To get a better idea, use the **-event** switch. This will generate a log of all of the communication events in the file '**bl**' in the current directory (this may be changed with the command line argument **-eventfile filename**). The program Upshot [8] may then be used to display the events. Use **tools.core/bin/$ARCH/upshot -l bl**. Look for large amounts of receive idle time (waiting for a message) and for long send and receive times (between the time a message is sent and when it is received). Possible ways to fix a performance problem here include

- switching to nonblocking sends and receives,
- reordering or rescheduling the communications, and
- adjusting the load balance.

5.4.3 Controlling the Event Log

While the default event log is suitable for many tasks, it is sometimes necessary to modify the content and formation of the event log. Two such modifications are discussed in this section; most readers should skip this section until they have used the event logs. The first modification is in the computation of the time that an event happens. On many computers, there is no single synchronized clock and Chameleon must synthesize one. As there is no single best way to do this, Chameleon provides for user-control of the synthesized clocks (by default, Chameleon uses a method that should be appropriate for most users). The second modification affects the output format of the event log, allowing the user to choose between two popular formats: Argonne's alog (used by **upshot**) and PICL (used by **ParaGraph**). It is also possible to add additional formats at runtime.

5.4.3.1 Controlling Event Clocks

Many parallel computer systems do not provide synchronized clocks. Chameleon contains code that attempts to synchronize the clocks using software, by estimating the difference between the local clocks. The default choice of synchronization method should be acceptable for most users; this section explains how these defaults may be modifed by the expert user.

The clock time is computed by taking a local time, computing an offset (shifting all of the clocks to the time of node zero), and adjusting for skew,

that is, the effect of clocks running at different rates. These adjustments can be modified by using the following arguments on the program's command-line following **-blogclock**:

noskew	Don't compute a modification of skew.
nooffsets	Don't compute a modification for offsets.
none	Don't compute any modification of local times.
print	Print out the values of offset and skew applied to each clock.

The opposite effect of the first two can be accomplished by removing the "**no**" in front of **noskew** or **nooffsets**.

Additional controls will be made available to the user soon, including control of the method used to determine the offsets and skews.

5.4.3.2 Controlling Output Format

The default output format is the Argonne alog format. This is a general event log format. Other formats can be generated; currently Chameleon also supports a subset of the (old) PICL trace format. This is the format that is needed for using the ParaGraph tools (available from *netlib*). This format does *not* support general user-defined events, but because of the utility of the ParaGraph system, it can be quite valuable in understanding program behavior.

The choice of output format can be changed with the command line parameter **-blogfmat**.

alog	Use alog format.
picl	Use (old) PICL format.

The default file name for alog files is '**bl**' and for picl is '**bl.trf**'.

One warning: the ParaGraph tools require a consistent clock (that is, they require that no message appear to arrive before it was sent). On systems without a single clock, this can be difficult to achieve. Using the **-blogclock** controls on systems without a single clock may or may not provide an accurate enough clock. In particular, the relatively low resolution of many workstation clocks may not be accurate enough for ParaGraph.

5.4.3.3 Load Balance

For the most efficient computation, all processors must be working all of the time. The more time that some processors spend waiting for others to complete their tasks, the less efficient a parallel computation will be. One place to look is at the amount of time spent waiting for a receive to start; this is idle time. Because it may require extra operations, Chameleon does not always provide

access to the time spent waiting for a receive to start. For p4 and PVM, it is necessary to use the command line option **-p4 busywait** (for p4) or **-pvm busywait** (for PVM) to get this information.

5.5 Runtime Control

It is sometimes necessary to control the collection of debugging information from within a program. For example, it is often helpful to collect an event log for only part of a program (such as for a particular routine whose performance needs to be investigated) rather than collect an event log for an entire program. The routine **PISetLogging** allows the programmer to selectively control the generation of event logs, trace files, and summary data. The format of this routine is

```
PISetLogging( level )
int level;
```

where **level** has the following bits:

1	Event logs
2	Tracing
4	Summary data

Using a value of 0 for level disables all options. The values may be "or"ed together to get the combinations. For example, a **level** of 3 turns on both event logs and tracing.

Chapter 6

Transport Layer-Specific Control

In most cases, completely portable programs can be written. However, in some cases, it is necessary to access features that are dependent on the message-passing system.

6.1 Specifying the Parallel Machines

In using a distributed network of machines as a parallel computer, it is necessary to specify which machines are to be used. In Chameleon, one can do this in several ways.

The simplest is to simply specify the number of machines; Chameleon will then use a file that lists available machines and attempt to find the requested number of machines. However, sometimes it is necessary to specify the specific machines. This can be done with the command line argument **-pihosts hostname[,hostname]**. For example, to run a program on the local machine and a workstation named **sparc1**, use **-pihosts sparc1**.

In some cases, even more information needs to be specified. In this case, a file should be used that contains the name of the machine and other relevant values. The format of this file is

```
# comments begin with a pound sign
machine-name [[exe=]name] [[np=]number] [wd=name] [arch=name]
```

where **ex** gives the name of the executable, **np** gives the number of processes on that machine, **wd** is the working directory to be used, and **arch** is the name of the architecture.

In addition, p4 users may specify a p4 procgroup file directly with the command line option **-pg filename** (or **-p4pg filename** for p4 version 1.3 and later).

6.2 Running Programs

Chameleon does not attempt to provide a common invocation environment, in large part because so many systems have special requirements. This section touches on some of the pecularities of these systems.

6.2.1 IBM EUI

The number of processors is taken from the environment variable **MP_PROCS**. If Ethernet is used for communications, the processors to use is taken from the file '**host.list**' in the current directory. Otherwise, there is (at this writing) no control over the choice of nodes (this is expected to change). Make sure that you use **COMM=eui** when invoking make. In addition, be sure that you do

```
setenv PWD '/bin/pwd'
```

immediately before executing the program if your login shell is the c-shell ('**/bin/csh**').

Some later versions of EUI (actually, the underlying support environment, POE) support the command-line argument **-procs np** for specifying the number of processors. This is known only to work from the Korn shell (**ksh**).

6.2.2 Intel Delta

Programs are started with an **mexec** command:

```
mexec -t "(nx,ny)" -f "program-name <arguments>"
```

where **nx,ny** is the size of the partition (for **nx * ny** total nodes) and **<arguments>** are the command line arguments for the program. Note the use of quotes; these are required.

6.2.3 TMC CM-5

CM-5 programs must be run from a front-end attached to a partition with a given number of nodes; no control over the number of processors is possible other than by chosing the partition to run in. Also, programs must be linked with a special linker; makefiles that use the compiler (**cc** or **f77**) to produce an executable will fail. Use **$(CLINKER)** or **$(FLINKER)** instead.

6.3 Command-Line Options

Several of the implementations provide for special options. Currently, p4 and PVM accept special options.

6.3.1 p4 Options

-p4 busywait Do a busywait during receives; this should be used only when using **-event** or **-summary** to get information on the time spent waiting for a receive to start.

-p4 list List all of the options available for p4.

6.3.2 PVM Options

-pvm vsnd Use virtual circuits (**vsnd** and **vrcv**) rather than the regular communication (**snd** and **rcv**). This is available only for PVM version 2.4.x.

-pvm busywait Do a busywait during receives; this should be used only when using **-event** or **-summary** to get information on the time spent waiting for a receive to start.

-pvm nointr Force PVM to disable interrupts during send and receive operations. This may be needed because PVM does not restart sends or receives that are interrupted. Use this if you get error messages about interrupted system calls.

-pvm list List all of the options available for PVM.

6.4 Setting Options

The command **PISetOption** allows the user to control system specific features. The format of this routine is

```
PISetOption( version, name, val )
char *version, *name;
void *val;
```

where **version** is the system name (such as **p4** or **pvm**), **name** is the name of the option, and **val** is a pointer to the data associated with **name**. For example,

```
PISetOption( "pvm", "vsnd", (void*)0) );
```

tells PVM to use **vsnd/vrcv** instead of **snd/rcv**.

Chapter 7

Reverse Compatibility

Chameleon provides a limited capability for running programs written using other message-passing systems. Currently, Chameleon supports running programs written using a subset of PICL or Intel NX.

7.1 PICL Compatibility

By linking with the appropriate file, many PICL programs can be run with few changes. The only change that you must make to the source code is to make the program hostless and use the `PICall` interface. Normally, this involves only a few simple changes to main program.

The other change is to insert the appropriate compatabilty interface module in the link line ahead of the Chameleon libraries. The name of the interface file is '`picl2comm$(COMM).a`', where `$(COMM)` is the usual `COMM` variable (equal either to "", "`eui`", "`p4`", or "`pvm`"). For example, this makefile fragment links the PICL program in '`pi.f`' with Chameleon:

```
TDIR = /usr/local/tools.core
LDIR = $(TDIR)/libs/libsg/$(ARCH)
pi: pi.f
       $(FLINKER) -g -o pi pi.f $(LDIR)/fmain.o \
                  $(LDIR)/picl2comm$(COMM).o \
                  $(TDIR)/fort/$(ARCH)/fort$(COMM).a \
                  $(TDIR)/fort/$(ARCH)/fort.a \
                  $(LDIR)/tools$(COMM).a $(LDIR)/tools.a \
                  $(LDIR)/system.a $(CLIB) $(SLIB)
```

As usual, symbols `CLIB` and `SLIB` are defined by the appropriate bmake include, and `ARCH` is the architecture.

Not all of PICL is supported. The supported routines include **send0**, **recv0**, **recvbegin0**, **recvend0**, **probe0**, **recvinfo0**, **who0**, **barrier0**, **sync0**, **bcast0**, **clock0**, **check0**, **gmax0**, **gmin0**, **gsum0**, and **open0**.

In addition, stubs (that do nothing but allow you to link an application) are provided for **setarc0**, **close0**, **clocksync0**, **tracelevel**, **tracenode**, **traceblockbegin**, **traceblockend**, **traceexit**, **tracefiles**, and **traceflush**.

For compatibility with codes written for the Intel iPSC/860, Delta, and Paragon, the Intel NX routines **mynode**, **mypid**, **mclock**, **gdlow**, and **gdhigh** are also supported.

One of the major features of PICL is its support of extensive tracing of operations. Chameleon provides a subset of those events. Use the command-line switches **-event -blogfmat picl** to produce a PICL-format trace file.

Versions on the IBM RS/6000 can support only Fortran or C, not both in the same program. This is because the names of the routines used by PICL create the same internal names from Fortran and C on the RS/6000 (on most other systems, the names are made different by adding an underscore or using upper case for Fortran and lower case for C). This choice is set when PETSc is installed; the default case is to support Fortran instead of C programs.

7.2 Intel NX Compatibility

By linking with the appropriate file, many Intel NX programs can be run with few changes. The only change that you must make to the source code is to make the program hostless and use the **PICall** interface. Normally, this involves only a few simple changes to main program.

The other change is to insert the appropriate compatabilty interface module in the link line ahead of the Chameleon libraries. The name of the interface file is '**nx2comm$(COMM).a**', where **$(COMM)** is the usual **COMM** variable (equal either to "", "**p4**", or "**pvm**"). For example, this makefile fragment links the NX program in '**pi.f**' with Chameleon:

```
LDIR = /usr/local/tools.core/libs/libsg/$(ARCH)
pi: pi.f
       $(FLINKER) -g -o pi pi.f $(LDIR)/nx2comm$(COMM).a \
                  $(LDIR)/tools$(COMM).a $(LDIR)/tools.a \
                  $(LDIR)/system.a $(CLIB) $(SLIB)
```

As usual, symbol **CLIB** is defined by the appropriate bmake include, and **ARCH** is the architecture.

Chameleon currently supports only a few special types of I/O operations, so the Intel NX I/O operations (e.g., **cwrite**, **iread**, and **restrictvol**) and the routines intended to support these (e.g., **eadd**) are not supported. The host/node model available on the iPSC/860 is also not supported.

The so-called forcetypes are supported by Chameleon. However, support is achieved by removing the forcetype bit from the tag; messages should have distinct tags in the lower 30 bits.

Chapter 8

Program Examples

This chapter contains some simple example programs.

8.1 Hello World

In the the classic "hello world" program, each processor writes "hello world from <node>" to standard output, where **<node>** is the number of each processor.

```
#include "tools.h"
#include "comm/comm.h"
int worker( argc, argv )
int  argc;
char **argv;
{
int i;
for (i=0; i<=PInumtids; i++)
    if (PIgtoken(PSAllProcs,i))
        printf( "Hello world from %d \n", PImytid );
return 0;
}
```

Note the use of **PIgtoken** to ensure that only one processor is writing at a time. The object file '**tools.core/libs/libsg/cmain.o**' is used to provide the main program; this uses the **PICall** interface to start the parallel program.

The makefile for this program is

```
ALL:  example1

ITOOLSDIR = /usr/local/tools.core

CFLAGS    = -I$(ITOOLSDIR) $(OPT) $(COPT)
```

```
LDIR      = $(ITOOLSDIR)/libs/libs$(BOPT)$(PROFILE)/$(ARCH)
LIBS      = $(LDIR)/cmain.o $(LDIR)/tools$(COMM).a $(LDIR)/tools.a \
            $(LDIR)/tools$(COMM).a $(LDIR)/system.a

LIBNAME = dummy

include $(ITOOLSDIR)/bmake/$(ARCH).$(COMM)
include $(ITOOLSDIR)/bmake/$(ARCH).$(BOPT)$(PROFILE)
include $(ITOOLSDIR)/bmake/$(ARCH)

example1: example1.o
        $(CLINKER) -o example1 $(CFLAGS) $(BASEOPT) example1.o \
                    $(LIBS) $(CLIB) $(SLIB) -lm
```

The command **make ARCH=intelnx BOPT=O** builds a version for Intel NX (i860, Delta, and Paragon) machines.

This same program written in Fortran is

```
      integer function worker()
      integer i, np
      include 'comm/fcomm.h'
c
      np = PInumtids()
      do 10 i=0,np
         if (PIgtoken(PSAllProcs,i) .ne. 0) then
              print *, "Hello world from ", PImytid()
         endif
10    continue
      worker = 0
      return
      end
```

8.2 Ring

A ring example circulates a value around a ring until it returns to the processor that started the ring. Note the use of the routine **PSNbrInRing** to compute the neighbor instead of the arithmetic expression **(PImyid + 1) % PInumtids**.

```
int worker( argc, argv )
int  argc;
char **argv;
{
int buf, siz = sizeof(int);
if (PImytid ==,0) {
    buf = 1;
    PIbsend( 1, &buf, siz, PSNbrRing(1,1,PSAllProcs), MSG_INT );
```

```
        PIbrecv( 1, &buf, siz, MSG_INT );
        }
    else {
        PIbrecv( 1, &buf, siz, MSG_INT );
        PIbsend( 1, &buf, siz, PSNbrRing(1,1,PSAllProcs), MSG_INT );
        }
    return 0;
    }
```

The execution of this program may be viewed by running it with the option **-event** and then running Upshot on the resulting log file. The sequential nature of this program will be obvious from the **upshot** display.

8.3 Rows and Columns

The following example shows the use of processor subsets to perform reductions along the rows and columns of a grid of processors. It also illustrates the use of the routine **SYArgGetInt** to find command line options (in this case, **-r rows** and **-c columns** for the size of the mesh of processors).

```
int worker( argc, argv )
int argc;
char **argv;
{
int     row, col, nrows, ncols, i;
double  vrow, vcol, work;
ProcSet *prow, *pcol;

/* Set the defaults for nrows and ncols */
nrows = 2;
ncols = 2;
PSSetMeshSize( PSAllProcs, nrows, ncols );
SYArgGetInt( &argc, argv, 1, "-r", &nrows );
SYArgGetInt( &argc, argv, 1, "-c", &ncols );
if (nrows * ncols != PInumtids) {
    if (PImytid == 0)
        fprintf( stderr,
                "[%d,%d] doesn't fit in %d processors \n",
                nrows, ncols, PInumtids );
    SYexitall("",1);
    }

PSMESHLOC( PSAllProcs, row, col );

/* note that first arguments must be distinct in different
```

```
   calls to PSPartition and greater than 0 and distinct for
   for the two processor sets. */
prow = PSPartition( row + 1, PSAllProcs );
pcol = PSPartition( col + nrows + 1, PSAllProcs );

vrow = vcol = (double) PImytid;
PIgdsum( &vrow, 1, &work, prow );
PIgdsum( &vcol, 1, &work, pcol );

for (i=0; i<=PInumtids; i++)
  if (PIgtoken(PSAllProcs,i))
     printf( "Row sum = %lf and column sum = %lf on %d \n",
             vrow, vcol, PImytid );
return 0;
}
```

8.4 Nonblocking Communication

The next example circulates a value around a ring until it returns to the processor that started the ring. This version uses nonblocking receives to allow the routines to compute while waiting for data.

```
int worker( argc, argv )
int  argc;
char **argv;
{
int        buf, siz = sizeof(int), tag = 1;
PIRecvId_t rid;
if (PImytid == 0) {
    buf = 1;
    PInrecv( tag, &buf, siz, MSG_INT, rid );
    PIbsend( tag, &buf, siz, 1, MSG_INT );
    PIwrecv( tag, &buf, siz, MSG_INT, rid );
    }
else {
    PInrecv( tag, &buf, siz, MSG_INT, rid );
    while (PInprobe( tag )) {
        /* Do some work while waiting for the message */
        }
    PIwrecv( tag, &buf, siz, MSG_INT, rid );
    PIbsend( tag, &buf, siz, PSNbrRing(1,PSAllProcs),
             MSG_INT );
    }
return 0;
}
```

Chapter 9

Installation

9.1 Code

The code may be acquired by anonymous ftp from **info.mcs.anl.gov** in directory '**pub/pdetools/chameleon.tar.Z**'. Fetch this file (using **type image**), uncompress it, and use tar to extract it. This will create a directory '**tools.core**'. The file '**tools.core/readme**' contains information on the installation process. For a quick start, set three environment variables:

TOOLSDIR	Directory containing tools.core
P4DIR	Directory containing p4, if you use p4
PVMDIR	Directory containing PVM, if you use PVM

Move into this directory and then execute '**bin/install**'. This will build the object libraries. (The command **bin/install -help** will detail the options that are available for the installation.)

9.2 Host File

For the workstation (p4 or PVM) versions to work, a list of available hosts (workstations and other computers) must be created. The format of this list is described in Section 2.3.3. Also, PVM or p4 must be installed. These may be found using **xnetlib**; p4 is also available by anonymous ftp from **info.mcs.anl.gov** in directory **pub/p4**.

Chapter 10

Summary of Routines

This chapter contains a brief summary of the routines in this manual. The chapter is organized into six major parts: program initiation, the point-to-point routines, the collective communication routines, the process set routines, environmental management, and parallel I/O routines. The beginning of each section lists the include files that are needed by C programmers. Fortran users should use the '**comm/fcomm.h**' file. If the word **MACRO** precedes the routine definition, it is a CPP macro for C users.

10.1 Program Initialization

```
#include "tools.h"
#include "comm/comm.h"
```

`int PICall( r, argc, argv )`
`int (*r)(), argc;`
`char **argv;`
	Calls a routine in a parallel execution mode.

10.2 Point-to-Point Routines

```
#include "tools.h"
#include "comm/comm.h"
```

`MACRO void PINewRecvBuf( msg, max,`
`type )`
`(type *)msg;`
`int max;`
	Allocates storage for receiving a message.

MACRO void PINewSendBuf(msg, max, type) (type *)msg; int max;	Allocates storage for sending a message.
MACRO void PIFreeRecvBuf(msg) void *msg;	Free storage for receiving a message
MACRO void PIFreeSendBuf(msg) void *msg;	Frees storage for sending a message.
void PISetNbrRoutines(tree, ring, mesh2d) int (*tree)(), (*ring)(), (*mesh2d)();	Sets the routines used to compute the neighbors.
void PISetOption(version, name, val) char *version, *name; void *val;	Sets a message-passing system-specific option.
MACRO void PIbprobe(type) int type;	Blocks until a message of a given type is available.
MACRO void PIbrecvProbed(type, buffer, length, datatype) int type, length, datatype; void *buffer;	Receives a message that has been probed.
MACRO void PIbrecvUnsz(type, buffer, length, datatype) int type, length, datatype; void *buffer;	Receives a message of unknown length.
MACRO void PIbrecvmrr(type, buffer, length, datatype) int type, length, datatype; void *buffer;	Receives a message from another processor.
MACRO void PIbrecvm(type, buffer, length, datatype) int type, length, datatype; void *buffer;	Receives a message from another processor.
MACRO void PIbrecvrr(type, buffer, length, datatype) int type, length, datatype; void *buffer;	Receives a message from another processor.
MACRO void PIbrecv(type, buffer, length, datatype) int type, length, datatype; void *buffer;	Receives a message from another processor.
MACRO void PIbsendmrr(type, buffer, length, to, datatype) int type, length, to, datatype; void *buffer;	Sends a message to another processor.

MACRO void PIbsendm(type, buffer, length, to, datatype) int type, length, to, datatype; void *buffer;	Sends a message to another processor.
MACRO void PIbsendrr(type, buffer, length, to, datatype) int type, length, to, datatype; void *buffer;	Sends a message to another processor.
MACRO void PIbsend(type, buffer, length, to, datatype) int type, length, to, datatype; void *buffer;	Sends a message to another processor.
MACRO void PIcrecv(id) PIRecvId_t id;	Cancels a previously issued PInrecv... .
MACRO void PIcsend(id) PISendId_t id;	Cancels a previously issued PInsend... .
MACRO int PIfrom()	Returns the processor that sent a received message.
MACRO int PInprobe(type) int type;	Tests whether a message of a given type is available.
MACRO void PInrecvmrr(type, buffer, length, datatype, id) int type, length, datatype; PIRecvId_t id; void *buffer;	Starts a nonblocking receive.
MACRO void PInrecvm(type, buffer, length, datatype, id) int type, length, datatype; PIRecvId_t id; void *buffer;	Starts a nonblocking receive.
MACRO void PInrecvrr(type, buffer, length, datatype, id) int type, length, datatype; PIRecvId_t id; void *buffer;	Starts a nonblocking receive.
MACRO void PInrecv(type, buffer, length, datatype, id) int type, length, datatype; PIRecvId_t id; void *buffer;	Starts a nonblocking receive.
MACRO void PInsendmrr(type, buffer, length, to, datatype, id) int type, length, to, datatype; PISendId_t id; void *buffer;	Starts a nonblocking send.

MACRO void PInsendm(type, buffer, length, to, datatype, id) int type, length, to, datatype; PISendId_t id; void *buffer;	Starts a nonblocking send.
MACRO void PInsendrr(type, buffer, length, to, datatype, id) int type, length, to, datatype; PISendId_t id; void *buffer;	Starts a nonblocking send.
MACRO void PInsend(type, buffer, length, to, datatype, id) int type, length, to, datatype; PISendId_t id; void *buffer;	Starts a nonblocking send.
MACRO int PInstatus(id) PISendId_t id;	Tests whether a nonblocking message has completed.
MACRO int PIsize()	Returns the length of a received message.
MACRO int PItype()	Returns the type of the most recently received message.
MACRO void PIwrecvmrr(type, buffer, length, datatype, id) int type, length, datatype; PIRecvId_t id; void *buffer;	Completes a nonblocking receive.
MACRO void PIwrecvm(type, buffer, length, datatype, id) int type, length, datatype; PIRecvId_t id; void *buffer;	Completes a nonblocking receive.
MACRO void PIwrecvrr(type, buffer, length, datatype, id) int type, length, datatype; PIRecvId_t id; void *buffer;	Completes a nonblocking receive.
MACRO void PIwrecv(type, buffer, length, datatype, id) int type, length, datatype; PIRecvId_t id; void *buffer;	Completes a nonblocking receive.
MACRO void PIwsendmrr(type, buffer, length, to, datatype, id) int type, length, to, datatype; PISendId_t id; void *buffer;	Completes a nonblocking send.

MACRO void PIwsendm(type, buffer, length, to, datatype, id) int type, length, to, datatype; PISendId_t id; void *buffer;	Completes a nonblocking send.
MACRO void PIwsendrr(type, buffer, length, to, datatype, id) int type, length, to, datatype; PISendId_t id; void *buffer;	Completes a nonblocking send.
MACRO void PIwsend(type, buffer, length, to, datatype, id) int type, length, to, datatype; PISendId_t id; void *buffer;	Completes a nonblocking send.

10.3 Collective Communication

```
#include "tools.h"
#include "comm/comm.h"
```

MACRO void PIbcastSrc(buf, siz, src, procset, datatype) void *buf; int siz, src, datatype; ProcSet *procset;	Broadcasts data to all processors.
MACRO void PIbcast(buf, siz, issrc, procset, datatype) void *buf; int siz, issrc, datatype; ProcSet *procset;	Broadcasts data to all processors.
MACRO void PIgcolx(lbuf, gsizes, gbuf, procset,datatype) void *lbuf, *gbuf; int *gsizes,datatype; ProcSet *procset;	Global collection from data of known size.
MACRO void PIgcol(lbuf, lsize, gbuf, gsiz, glen, procset, datatype) void *lbuf, *gbuf; int lsize, gsiz, *glen, datatype; ProcSet *procset;	Global collection from data of unknown size.

MACRO void PIgcmax(val, n, work, procset) char *val, *work; int n; ProcSet *procset;	Computes global maximum reduction.
MACRO void PIgcmin(val, n, work, procset) char *val, *work; int n; ProcSet *procset;	Computes global minimum reduction.
MACRO void PIgcsum(val, n, work, procset) char *val, *work; int n; ProcSet *procset;	Computes global sum reduction.
MACRO void PIgdmax(val, n, work, procset) double *val, *work; int n; ProcSet *procset;	Computes global maximum reduction.
MACRO void PIgdmin(val, n, work, procset) double *val, *work; int n; ProcSet *procset;	Computes global minimum reduction.
MACRO void PIgdsum(val, n, work, procset) double *val, *work; int n; ProcSet *procset;	Computes global sum reduction.
MACRO void PIgfmax(val, n, work, procset) float *val, *work; int n; ProcSet *procset;	Computes global maximum reduction.
MACRO void PIgfmin(val, n, work, procset) float *val, *work; int n; ProcSet *procset;	Computes global minimum reduction.
MACRO void PIgfsum(val, n, work, procset) float *val, *work; int n; ProcSet *procset;	Computes global sum reduction.

MACRO void PIgimax(val, n, work, procset) int *val, n, *work; ProcSet *procset;	Computes global maximum reduction.
MACRO void PIgimin(val, n, work, procset) int *val, n, *work; ProcSet *procset;	Computes global minimum reduction.
MACRO void PIgisum(val, n, work, procset) int *val, n, *work; ProcSet *procset;	Computes global sum reduction.
MACRO void PIgsync(procset) ProcSet *procset;	Synchronizes processors.
MACRO int PIgtoken(procset, i) ProcSet *procset; int i;	Passes a "token" among processors.
void PISetCollectionFunc(func) void (*func)();	Sets the function use for collections (PIgcol).
void PISetCombFunc(func) void (*func)();	Sets the function use for reductions (PIgdsum, etc.).
void PISetScatterFunc(func, funcsrc) void (*func)(), (*funcsrc)();	Sets the function use for scatters (PIbcast).
void PISetSyncFunc(func) void (*func)();	Sets the function use for synchronizations (PIgsync).

10.4 Process Set Management

```
#include "tools.h"
#include "comm/comm.h"
```

void PSAddMember(procset, p, np) ProcSet *procset; int *p, np;	Adds one or more processors as members of processor set.
void PSCompile(procset) ProcSet *procset;	Compiles a processor set.
ProcSet *PSCreate(name) int name;	Creates a processor set structure.
void PSDestroy(procset) ProcSet *procset;	Destroys a processor set structure.

`ProcSet *PSPartition( pval, procset )`
`int pval;`
`ProcSet *procset;`
	Computes a partition dynamically by using an id value to partition processors into disjoint sets.

`void PSPrintProcset( ps, form, fp )`
`ProcSet *ps;`
`int form;`
`FILE *fp;`
	Displays a processor set in a simplified fashion.

`ProcSet *PSUnion( ps1, ps2, name )`
`ProcSet *ps1, *ps2;`
`int name;`
	Forms a procset from the union of two procsets.

`MACRO int PSISROOT(procset)` `ProcSet *procset;`	Returns 1 if this processor is the root of the procset, 0 otherwise.

`MACRO void PSMESHLOC( procset, i, j )`
`ProcSet *procset;`
`int *i, *j;`
	Returns the location of the processor in the mesh.

`MACRO int PSMYPROCID( procset )` `ProcSet *procset;`	Returns the relative processor number in a processor set.
`MACRO int PSNUMNODES( procset )` `ProcSet *procset;`	Returns the number of nodes in a processor set.

`int PSNbrMesh( offx, offy, wrapx,`
`wrapy, procset )`
`int offx, offy, wrapx, wrapy;`
`ProcSet *procset;`
	Returns the processor id's of neighbors in a mesh.

`int PSNbrRing( offset, wrap, procset )`
`int offset, wrap;`
`ProcSet *procset;`
	Returns the processor id's of neighbors in a ring.

`int PSNbrTree( nbr, procset )`
`PS_Tree_t nbr;`
`ProcSet *procset;`
	Returns the selected child or parent of this node.

`MACRO void PSPROCLIST(procset,list)`
`ProcSet *procset;`
`int *list;`
	Returns the processors in the processor set. This is the ordering used by PIgcolx.

`MACRO int PSROOT(procset)` `ProcSet *procset;`	Returns global id of the root processor of this processor set.

`void PSSetMeshSize( procset, nx, ny )`
`ProcSet *procset;`
`int nx, ny;`
	Sets the size of the mesh to use.

10.5 Environmental Management

```
#include "tools.h"
#include "comm/comm.h"
```

`MACRO int PIdiameter`	Returns the maximum number of hops between two processors.
`MACRO int PIdistance( from, to)` `int from, to;`	Returns the number of hops between two processors.
`MACRO int PImytid`	Returns processor id of calling processor.
`MACRO int PInumtids`	Returns the number of processors.
`int PIGetNbrs( myid, nbrs )` `int myid, *nbrs;`	Returns all of the immediate neighbors of a node.

`int PIGetTypes( procset, n )`
`ProcSet *procset;`
`int n;`
	Gets a range of message types that are unique to a processor set.

`MACRO void PIMsgSizes( min, max)` `int *min, *max;`	Returns the range of message sizes.

`void PINodeName( name, maxlen )`
`char *name;`
`int maxlen;`
	Creates a string containing the name of the node.

`void PISetLoggingBit( bit, flag )` `int bit, flag;`	Sets/clears the logging level for communications, for a single option.
`void PISetLogging( level )` `int level;`	Sets the logging level for communications.
`void PISetMergeEventFiles( flag )` `int flag;`	Controls whether event files are merged.
`void PISetPacketSize( val )` `int val;`	Sets the packet size for collective operations.
`void PISetRRSize( val )` `int val;`	Sets the size for use of ready-receiver (force) in collective operations.
`void PISetTracefile( name )` `char *name;`	Sets the name of the file for tracing output.
`MACRO void PITagRange( low, high)` `int *low, *high;`	Returns the range of value (user) message tags.

10.6 I/O Routines

```
#include "tools.h"
#include "comm/comm.h"
```

`void PIFPrintArraySpec( fp, sz, nd )`
`FILE *fp;`
`PIArrayPart *sz;`
`int nd;`
	Prints a distributed array specifier.

`void PIFclose( fp )` `PIFILE *fp;`	Closes a parallel file.
`void PIFflush( fp )` `PIFILE *fp;`	Flushes the output to a parallel file.

`PIFILE *PIFopen( name, procset, mode, pmode )`
`char *name;`
`ProcSet *procset;`
`int mode, pmode;`
	Opens a parallel file.

`void PIReadCommon( fp, fmat, flen, v, n, datatype )`
`PIFILE *fp;`
`char *fmat;`
`void *v;`
`int flen, n, datatype;`
	Reads data from a parallel file.

`void PIReadDistributedArray( fp, fmat, flen, sz, nd, v, datatype )`
`PIFILE *fp;`
`char *fmat;`
`void *v;`
`PIArrayPart *sz;`
`int flen, nd, datatype;`
	Reads a distributed array from a parallel file. The EXACT same data will be read in independent of the number of processors.

`void PIWriteCommon( fp, fmat, flen, v, n, datatype )`
`PIFILE *fp;`
`char *fmat;`
`void *v;`
`int flen, n, datatype;`
	Writes data to a parallel file.

`void PIWriteDistributedArray( fp, fmat, flen, sz, nd, v, datatype )`
`PIFILE *fp;`
`char *fmat;`
`void *v;`
`PIArrayPart *sz;`
`int flen, nd, datatype;`
	Writes a distributed array to a parallel file.

Acknowledgments

The work described in this report has benefited from conversations with and use by a large number of people. Among the contributers are David Levine, who read the drafts, and the early users who requested additional functionality and were patient with our bug fixes. Interaction with the MPI committee helped the organization of this manual by emphasizing the need to provide an easy-to-use subset.

Bibliography

[1] A. Beguelin, J. Dongarra, G. A. Geist, R. Manchek, and V. Sunderam. A user's guide to PVM: Parallel virtual machine. Technical Report TM-11826, Oak Ridge National Laboratory, 1991.

[2] Ralph Butler and Ewing Lusk. User's guide to the p4 parallel programming system. Technical Report ANL-92/17, Argonne National Laboratory, October 1992.

[3] Ian Foster, William Gropp, and Rick Stevens. The parallel scalability of the spectral transform method. *Monthly Weather Review*, 120:835–850, 1992.

[4] G. A. Geist, Michael T. Heath, B. W. Peyton, and Patrick H. Worley. PICL: A portable instrumented communications library. Technical Report TM-11130, Oak Ridge National Laboratory, 1990.

[5] William Gropp and Edward Smith. Computational fluid dynamics on parallel processors. *Computers and Fluids*, 18:289–304, 1990.

[6] William D. Gropp and Ewing Lusk. A test implementation of the MPI draft message-passing standard. Technical Report ANL-92/47, Argonne National Laboratory, December 1992.

[7] Michael T. Heath and Jennifer Etheridge Finger. Visualizing performance of parallel programs. *IEEE Software*, 8(5):29–39, September 1991.

[8] Virginia Herrarte and Ewing Lusk. Studying parallel program behavior with Upshot. Technical Report ANL-91/15, Argonne National Laboratory, August 1991.

[9] Mark T. Jones and Paul E. Plassmann. An efficient parallel iterative solver for large sparse linear systems. In *Proceedings of the IMA Workshop on Sparse Matrix Computations: Graph Theory Issues & Algorithms*, Minneapolis, 1991. University of Minnesota.

Function Index

Distribution for ANL-93/23

Internal:

J. M. Beumer (100)
F. Y. Fradin
W. D. Gropp (25)
G. W. Pieper
R. L. Stevens
C. L. Wilkinson
TIS File

External:

DOE-OSTI, for distribution per UC-405 (54)
ANL-E Library (2)
ANL-W Library
Manager, Chicago Operations Office, DOE
Mathematics and Computer Science Division Review Committee:
W. W. Bledsoe, The University of Texas, Austin
B. L. Buzbee, National Center for Atmospheric Research
J. G. Glimm, State University of New York at Stony Brook
M. T. Heath, University of Illinois, Urbana
E. F. Infante, University of Minnesota
D. O'Leary, University of Maryland
R. E. O'Malley, Rensselaer Polytechnic Institute
M. H. Schultz, Yale University
J. Cavallini, Department of Energy - Energy Research
F. Howes, Department of Energy - Energy Research
B. Smith, University of California, Los Angeles (15)

END

DATE FILMED

12/6/93