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

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

ANL-93/00

Chameleon
Parallel Programming Tools
User's Manual

Draft

by

William Gropp
Mathematics and Computer Science Division

Barry Smith
Department of Mathematics
University of California, Los Angeles

February 1993

This work was supported in part by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38.

Contents

Chameleon
Parallel Programming Tools
User's Manual

Draft

by

William Gropp
Mathematics and Computer Science Division
Argonne National Laboratory

Barry Smith
Department of Mathematics
University of California, Los Angeles

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 such system. Rather than replacing these existing systems, Chameleon is meant to supplement them. 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 is tracking the MPI (Message Passing Interface) 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 and Thinking Machines CMMD (CM-5).

Chapter 1

Summary

This chapter contains a short summary that will get you started with Chameleon. It does not contain all of the routines in all of their glory; there is just enough to get you started. The calling sequences for these routines can be found on their man pages; example programs are in '`tools.core/comm/examples`'.

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, f, etc.)
PIgXmax	Form the global maximum of a vector
PIgXsum	Form the global sum of a vector
PIbcast	Broadcase 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 6.

Chapter 2

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 provides 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) will run *unchanged* on a collection of workstations as well as on parallel supercomputers such as the Intel Delta and Thinking Machines CM-5. Another feature is that a wide variety of debugging (both for correctness and for performance) information is made available. A production library that imposes *no* overhead is also provided (again requiring no change in the source code).

This 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.

2.1 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.

2.2 Organization of the manual

This manual is organized roughly into four sections. Chapter 3 describes how programs are initialize Chameleon. This chapter contains information on both homongenous MPP parallelism and heterorgeneous, disributed workstation parallelism. Chapter 4 describes point-to-point message passing routines. Chapter 5 describes the routines for collective operations, including the routines for define subsets of processes. The final Chapters describe some special topics, such as debugging (Chapter 6) 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 2.6 explains how to write a `makefile` to use with Chameleon.

2.3 Systems supported

Chameleon supports both native (vendor) communications libraries and several popular "portable" communications packages. The "portable" packages supported are P4 [2], PICL [3], 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 and Touchstone Delta, and for the CM-5, using version 2 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 has the advantage that using a host can seriously 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 this manual, may be used to restrict operations to the remaining nodes.

There is also "inverse" support for some of the "portable communications" systems. Programs using the supported subsets of PICL 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 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.

2.4 Languages supported

The C language is the primary language that is supported. Extensive use is made of the C preprocessor to provide many of the functions, 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. 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.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). If you are interested in how this is accomplished, look in the directory '`tools.core/c2fort`'. The program in this directory uses the same approach that is used to generate the manual pages from the C source files.

2.5 Programming

Most 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 `-Itools.core`. (It is necessary to include this because these include files may include additional include files; the path names for these additional include files is relative to the root of the tools tree.)

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

2.6 Linking

To build programs with Chameleon you need to link with a number of libraries. To simplify the use of 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/libsOpg`'

production '`tools.core/libs/libsO`'

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 Sun4 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 sun4's:

```
COMM = p4
BOPT = O
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 sun4. The `include` lines include 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).

2.7 Chameleon and message-passing standards

There are a large number of message-passing systems available. In addition, there is an effort to develop a message-passing standard, refered to as the MPI (Message Passing Interface). 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 the message-passing system; 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. It is the hope of the developers of Chameleon that it can be replaced by the MPI standard; until that standard is specified and widely available, 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 addition, 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).

2.8 Further Information

refmanual

More detailed information about the routines mentioned in this manual may be found in the man pages (using `toolman`). The script `toolman` is one of the tools provided by PETSc for accessing the detailed documentation on the routines. 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.

For programs using simple decompositions of regular arrays, the manual "BlockComm for Fortran" [?] should be consulted.

Chameleon is continually growing through the addition of new routines. Suggestions (and bug reports) should be emailed to '`gropp@mcs.anl.gov`'.

Additional documents that may be of interest include "BlockComm" (a package for block-oriented message passing), and "Tools Overview" (an overview of the Tools package).

Chapter 3

Starting Programs

One of the least considered but most important areas 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 these 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.

In practice, this is done by replacing the code

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

with

```
main(argc,argv)
int  argc;
char **argv;
{
int main2();
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.

An even simpler interface is available; just replace

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

with

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

This uses a main program provided by the file '`cmain.o`' in the tools library (for example, '`tools.core/libs/libsg/sun4/cmain.o`'). (This main program simply calls `worker` using `PICall`; all of the discussion on how `PICall` handles starting a program and processing argument lists applies to this interface as well.)

3.1 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 6. Some of these are meaningful only for distributed computing (such as on a collection of workstations).

3.1.1 Parallelism arguments

These describe the number of processes, location of the executable, and the type of machine to run the program on. The arguments are:

-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

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 Sun4, the architecture will be Sun4 and the name of the executable will be that of the running program). These arguments may be used multiple times to specify multiple architectures. For example, to use both sun4's and rs6000's, specify

```
-arch sun4 -arch rs6000
```

Specific numbers of each may be specified by using `-np`. For example, 5 sun4's and 8 rs6000's can be specified by

```
-np 5 -arch sun4 -np 8 -arch rs6000
```

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.

3.1.2 Resource Limits

With any program, it is necessary to set some limits on the amount of resource 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 run-away 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, and

-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`.

3.1.3 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 many 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 what 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, principle 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, these three lines indicate that the machine `mysun` is available for small jobs from 8am to 6:30pm Monday through Friday and 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 server with 20 processors. This machine is a server, 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 location of the hosts file is given by the environment variable `TOOLSHOSTS`; 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 principle 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.

3.1.4 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 the chapter on debugging.

-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.

3.2 In case of trouble

On a collection of workstations, the most common cause of trouble is not having enough workstations available. `PICall` uses a database of workstations (described above) 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 data base, 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 database file is '`tools.core/comm/hosts`'. The file name can be overridden by specifying the environment variable `TOOLSHOSTS`.

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.

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. 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 will significantly reduce the time that it takes to start a parallel application.

If all of this fails, look at Chapter 6. The options `-trace` may be used to indicate where a program is getting hung.

Chapter 4

Message Passing

Message passing is a well-known and portable method for writing parallel programs. This document will 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 if there are any higher-level communication or computation routines (such as a parallel linear solver) that 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 is possible). This means that 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.

4.1 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).

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

<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 Non-blocking operation. When the routine completes, a handle has been set. It is necessary to use the `wait` modifier with this handel to insure that the operation (both sends and receives) has completed.

w For non-blocking 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`).

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.

It is not necessary to master all of these routines to write a working parallel program. To begin with, use the blocking message routines with user buffers. These 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 meaning of the parameters in these routines is descibed below.

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

4.1.1 Data types

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:

MSG_SHRT short (short integer)

MSG_INT integer

MSG_LNG long (long integer)

MSG_FLT float (floating point)

MSG_DBL double (floating point)

MSG_OTHER unspecified, can be used for char and for any data whose format is unspecified.

This raises the issue of what to do if a message containing different data types 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 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.) For the most general case, you may use the methods described in section 4.7 for sending and receiving messages containing different data types.

4.2 Blocking messages

Blocking messages are 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

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 Receives 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 (int) message tag.

buffer (char *) pointer to buffer

datatype (int) datatype (e.g., `MSG_INT`)

length (int) size of message to send in bytes

maxlength (int) size of the buffer in a receive (allows a message of this size or smaller) in bytes

size (int) size of a received message in `RECVSYNCUNSZ` in bytes.

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

The variants such as `PIbrecv` and `PIbsendrr` have the same calling sequences but the slightly different semantics as described in the overview. There are six such routines:

PIbrecv PIbrecvrr PIbrecvmrr
PIbsend PIbsendrr PIbsendmrr

4.3 Nonblocking messages

Nonblocking messages allow 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 `PIwsend`s.

In the case of a nonblocking receive, the programmer is telling 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

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

(The reason that the “wait” has the same parameter list as the “non-blocking” operation is to make it easy to simulate non-blocking operations with blocking ones. It also aids the programmer in seeing just what data was expected at the end of a non-blocking 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:

PInrecv PInrecvrr PInrecvmrr
PInsend PInsendrr PInsendmrr
PIwrecv PIwrecvrr PIwrecvmrr
PIwsendPIwsendrrPIwsendmrr

In addition to these routines, the routine `PInstatus` may be used to determine if 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. This may be used to select different algorithms in the two cases at compile time.

4.4 Message data

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 was. These are available from the 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 what 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 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 if 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:

`PInprobe(tag)`	Non-blocking 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`. This is needed because of race conditions in some implementations of `PIbprobe` and `PInprobe`. The format of `PIbrecvProbed` is

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

4.5 Message buffers

For the best performance, 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 `NOMEM` versions of the send and receive routines. (Fortran users can not allocate memory this way, so they can not 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 `double`s, 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`.

4.6 Message topology

Most 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. 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.

4.7 Non-contiguous messages

The message-passing routines described above are sufficient to write relatively efficient programs. In many applications, however, the data to be sent or received is not in consequtive locations in memory. While any data, however laid out in memory, may be sent or received this way, there are a number of special cases for which special arrangements may be made by the operating system or the hardware. These include constant, non-unit strides (useful for sending a row from a Fortran array), scatter/gathers (useful for sparse vectors), and non-contiguous blocks (useful for sending structures with pointers). These operations may be provided by the low-level hardware and software because they are common and because important performance optimizations can be made for them, particularly the elimination of unnecessary memory motion. For Unix programmers, the non-contiguous block case is simply the analogue of the routines `readv` and `writev`. Several message-passing implementations already provide non-unit stride vector operations, and research work with scatter/gather operations [5] has shown that these are also of great potential benefit.

Chameleon will provide access to all of these in any combination by way of a general data-arrangement descriptor. (This is the approach taken by the MPI draft standard.) If the native system does not provide them, they are emulated by Chameleon (by using a buffer local to the routine, and packing or unpacking that buffer).

Note that these routines are not yet implemented.

4.7.1 Organization

These routines are organized much as the "contiguous" ones are. The difference in naming is only the insertion of an additional name to indicate an arbitray data-arrangement (`v`). Because of the description of the data (buffer) arrangement is made by a single parameter, the calling sequences of these routines are slightly different than those for contiquous data buffers.

A data-arrangement is defined in 2 steps: (1) create a data-arrangement descriptor and (2) add descriptions of the data arrangement to it. A data-arrangement descriptor is created with `PINewDA`. The routines `PIDAVec`, `PIDABlk`, and `PIDAIdx` add descriptions for non-unit stride vectors, blocks, and indexed vectors (these are described in more detail on the man pages for these routines). A send or receive operation then takes the data-arrangement descriptor instead of the (buffer,size) that the send and receive routines use for contiguous data.

4.7.2 Native implementations

The current plan is to use native (vendor) implementations for the following transport layers (that is, these are not emulated by Chameleon):

vectors PVM, CM-5, *nx/Paragon*

index PVM

blocks PVM, *nx/Paragon*

(The *nx/Paragon* implementation will be available when the nx/Paragon is.)

PVM is included here because the design of PVM encourages the contstruction of messages consisting of many parts. PVM itself does not provide these operations directly.

4.7.3 Vectors

The "vector" data-arrangement allows sending and receiving of non-contiguous data where the elements are separated by a constant distance or stride.

To add a vector to the data arrangement, use

```
PIDAVec( da, buffer, elmlen, stride, elmcount, datatype )
```

The parameters are:

buffer (char *) pointer to buffer

elmlen size of each element, in bytes

stride distance between successive elements, in bytes

elmcount number of elements

datatype (int) datatype (`MSG_INT`)

For example, if a Fortran array is declared as

```
double precision a(10,20)
```

and the row `a(3,1:20)` is to be sent, use

```
da = PIDANew(1)
PIDAVec(da,a(3,1),8,8*10,20)
PIbvsend(da,10,3)
```

Here we assume that a double precision quantity takes eight bytes.

4.7.4 Blocks

Blocks of contiguous data may be added to the data arrangement with `PIDABlk`. The format is

```
PIDABlk( da, buffer, len, datatype )
```

Multiple uses of `PIDABlk` may be used to send complicated data structures. For example, to send the structure

```
struct {
    int n[3];
    double a[4];
    char   s;
    } S;
```

we can use the following code:

```
da = PIDANew( 3 );
PIDABlk( da, &S.n[0], 3 * sizeof(int),    MSG_INT );
PIDABlk( da, &S.a,    4 * sizeof(double), MSG_DBL );
PIDABlk( da, &S.s,    sizeof(char),       MSG_OTHER );
PIbvsend( da, tag );
```

To receive the structure, the exact same code is used, with the `PIbvsend` replaced with

```
PIbvrecv(da,tag);
```

4.7.5 Indexed

Another type of non-contiguous communication pattern that is common in applications is "indexed." For example, if elements of one vector, indexed by another, are used. These are defined with `PIDAIdx`. The format is

```
PIDAIdx( da, buffer, idxarray, numitems, datatype )
```

Chapter 5

Collective operations

Many programs require operations that involve a collection of processes. When every processor is involved, these are often 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. To use these routines, create a processor subset 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 `PIAllProcs`.

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 is deleted. As a practical matter, using the 3-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 may include such things as choosing special communications schedules for the collective operations.

5.1 Processor sets

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.

```
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 additional 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).

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

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 `PIAllProcs` is the processor set containing all of the nodes in the parallel machine.

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

5.1.1 Information on a subset

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.

PSNUMNODES The routine `PSNUMNODES(procset)` returns the number of processors in the processor set.

PSMYPROCID The routine `PSMYPROCID(procset)` returns the *relative* id of the processor in the processor set. This value is between 0 and `PSNUMNODES(procset)`-1.

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( PIAllProcs )` is equivalent to `0`.

PSSetMeshSize The routine `PSSetMeshSize(procset,nx,ny)` specifies the dimensions of the processor set when considered as a 2-dimensional mesh.

PSMESHLOC The routine `PSMESHLOC(procset,i,j)` sets `i` and `j` to the relative location of the processor in a 2-dimensional mesh (as defined by `PSSetMeshSize`).

PSNbrRing Returns the processor ids of neighbors in the processor set when considered as a 1-dimensional ring. See the man page for more details.

PSNbrMesh Returns the processor ids of neighbors in the processor set when considered as a 2-dimensional mesh. See the man page for more details.

PSNbrTree Returns the processor ids of neighbors in the processor set when considered as a binary tree. See the man page for more details.

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.

5.2 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 together, it is necessary to know what data type and format is being combined. These data types 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

`PIgXand` Finds the logical and of the values across the processor set

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 maxval, work;
...
PIgdmax( &maxval, 1, &work, PIAllProcs );
```

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

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

5.2.1 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`, and

op routine to combine values together.

The format of `PISetCombFunc` is

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

5.3 Scatter/gather

Occasionally, it is necessary to either scatter data from one processor to many others, or to collect data from all processors. These operations are supported by scatter and gather operations.

5.3.1 Broadcast

These 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 sets 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, non-disjoint processor sets, all of which are using a `PIbcast`, it is necessary to insure that there is no possibility of deadlock.

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`, and

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)();
```

5.3.2 Gather

These 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 Holds the local contribution,

lsize Number of bytes in the lbuf,

gbuf Buffer to hold the result,

gsize Size of gbuf in bytes,

glen The actual number of bytes received is returned here,

procset Processor set to collect over, and

datatype data type 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 Holds the local contribution

gbuf Buffer to hold the result

gsizes Array of the sizes of each contribution. `gsizes[PImytid]` is the size of the local contribution.

procset Processor set to collect over

datatype data type of lbuf and gbuf.

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

5.4 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 insure that only one writes at a time. A simple way to do this is with a barrier:

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

However, this can be very inefficient. For this kind of use, the routine `PIgtoken` provides an alternate approach. The same effect as above is achieved with

```
for (i=0; i<=PInumtids; i++) {
    if (PIgtoken(PIAllProcs,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`.

Chapter 6

Debugging

Debugging a parallel program can be difficult. Many of the most powerful debugging aids for uniprocessors (such as `dbx`) are either 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 tools provided by Chameleon to aid in debugging a program, both for correctness and performance.

6.1 Correctness

There are two main results of errors in message-passing parallel programs that are not present in uniprocessor programs. These are deadlock (program "hangs") and non-deterministic behavior (different results for the same input data in different runs). Both of these are often caused by the program 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 occured. 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 if 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, then use `diff` to find the first place that the logs differ.

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

```
[0] Starting <PIgscatter> [args.c:93]
[1] Starting <PIgscatter> [args.c:93]
[0] Ending <PIgscatter> [args.c:93]
[0] recvstart <Tag 1> [buggy.c:17]
[1] Ending <PIgscatter> [args.c:93]
[1] recvstart <Tag 1> [buggy.c:17]
```

(This is from the ring example below, but with all processors executing their `PIbrecv` before the `PIbsend`.) Note the `PIgscatter`s from the file '`args.c`'; these are communication operations issued by `PICall` as part of the program startup. The two lines from file '`buggy.c`' at line 17 are the begining 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 convienently 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 the cshell script

```
foreach file (log.*)
    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 the operating system (fails to) deal with multiple processes writing to the same file.

6.2 Error messages

The debugging version of the PETSc 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).

6.3 Performance

Performance debugging can be very difficult. There are many sources of inefficiency in a parallel program 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. To get these machine-dependent parameters, the program `twin` in '`comm/examples/angst`' can be run.

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`'.

6.3.1 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 to 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. The routines `SYGetCPUTime`, `SYusc_clock`, and `SYuscDiff` may be used.

6.3.2 Communication Implementation

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

- long transit times
- long startups
- 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
Send:            9        112 1.522400e-02
Recv:            9        112 6.892758e+00
Global:         15          0 6.913982e+00
```

Look for

- low transfer rates but long messages
- 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 '`blog.0`' in the current directory (this may be changed with the command line argument `-eventfile filename`). The program `upshot` [4] may then be used to display the events. Use `tools.core/bin/$ARCH/upshot -l blog.0`. 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

- Switch to nonblocking sends and receives
- Reorder or reschedule the communications
- Adjust the load balance

6.3.2.1 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 adjustmest 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 these 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.

6.3.3 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.

6.4 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 some 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 7

System-specific control

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

7.1 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, this can be done 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 [[ex=]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.

7.2 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 8

Program Examples

This chapter contains some simple example programs.

8.1 Hello world

This is the classic "hello world" program. In this case, each processor writes "hello world" to standard output.

```
#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(PIAllProcs,i))
        printf( "Hello world from %d\n", PImytid );
return 0;
}
```

Note the use of `PIgtoken` to insure 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 startup the parallel program.

8.2 Ring

This example circulates a value around a ring until it returns to the processor that started the ring.

```
int worker( argc, argv )
int  argc;
char **argv;
{
int buf, siz = sizeof(int);
if (PImytid == 0) {
    buf = 1;
    PIbsend( 1, &buf, siz, 1, MSG_INT );
    PIbrecv( 1, &buf, siz, MSG_INT );
    }
else {
    PIbrecv( 1, &buf, siz, MSG_INT );
    PIbsend( 1, &buf, siz,
                   (PImytid + 1) % PInumtids, 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

This 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;
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);
    }

row = PImytid % nrows;
col = (PImytid / nrows);

/* note that first arguments must be distinct in different
   calls to PSPartition and greater than 0 */
prow = PSPartition( row + 1, PIAllProcs );
pcol = PSPartition( col + nrows + 1, PIAllProcs );

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

for (i=0; i<=PInumtids; i++)
  if (PIgtoken(PIAllProcs,i))
     printf( "Row sum = %lf and column sum = %lf on %d\n",
               vrow, vcol, PImytid );
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`'. Next, 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 3.1.3. 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`.

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, Mathematics and Computer Science Division, October 1992.

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

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

[5] Matt Rosing and Joel Saltz. Low latency messages on distributed memory multiprocessors. manuscript, 1992.

Function Index

P

S