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

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

ANL-92/17

User's Guide to the p4 Programming System

by

Ralph Butler and Ewing Lusk*

Mathematics and Computer Science Division

October 1992

This work was supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research under Contract W-31-109-Eng-38.

*Also of The University of North Florida, Department of Computer Science, Jacksonville, Florida

Contents

1 Introduction 1

2 Structure of the Distribution Directory 2

3 Installing p4 3

3.1 Installing the p4 System 3

3.2 Installing the Documentation 4

3.3 Examples included with the Distribution 5

4 Getting Started 5

4.1 A Message-Passing Example 5

4.2 Program Description 5

4.3 Analysis of the Program 5

5 Specifying Processes in the Procgroup File 6

6 Developing a Simple p4 Program 8

6.1 A Minimal Example 8

6.2 A More Complicated Example 9

7 Command-Line Arguments 12

8 The p4 Function Library 12

8.1 Overview of the Library 12

8.2 Return Codes from p4 Functions 13

9 p4 Functions for Managing Processes and Clusters 13

9.1 Functions for Process Management 13

9.2 Functions for Cluster Management 15

10 Functions for Message Passing 15

10.1 Explicit Sending and Receiving of Messages 16

10.2 Global Operations 17

11 Functions for Shared Memory **19**
11.1 Managing Shared and Local Memory 19
11.2 Shared Memory Data Types 19
11.3 Monitor-Building Primitives 19
11.4 Some Useful Monitors 20

12 Functions for Timing p4 Programs **22**

13 Functions for Debugging p4 Programs **23**

14 Miscellaneous Functions **25**

15 Fortran Interface **26**

16 Faster Startup with the Secure Server **27**

17 Utilities for Managing a p4 Session **28**

18 Creating Logfiles for Upshot **28**
18.1 User-Specified Events 29
18.2 Examining Log Files with Upshot 30
18.3 Automatic Logging of p4 Events 31

19 Machine-Specific Notes **31**

20 Some Common Problems and their Solutions **33**

21 Concept Index **35**

22 Function Index **35**

1 Introduction

P4 is a library of macros and subroutines developed at Argonne National Laboratory for programming a variety of parallel machines. Its predecessor was the m4-based "Argonne macros" system described in the Holt, Rinehart, and Winston book *Portable Programs for Parallel Processors*, by Lusk, Overbeek, et al., from which p4 takes its name[1]. The current p4 system maintains the same basic computational models described there (monitors for the shared-memory model, message-passing for the distributed-memory model, and support for combining the two models) while significantly increasing ease and flexibility of use. See 4 [Getting Started], page 5 for a simple example.

P4 is intended to be portable, simple to install and use, and efficient. It can be used to program networks of workstations, advanced parallel supercomputers like the Intel Touchstone Delta and the Alliant Campus HiPPI-based system, and single shared-memory multiprocessors. It has currently been installed on the following list of machines: Sequent Symmetry (Dynix and PTX), Encore Multimax, Alliant FX/8, FX/800, and FX/2800, Cray X/MP, Sun, NeXT, DEC, Silicon Graphics, HP, and IBM RS6000 workstations, Stardent Titan, BBN GP-1000 and TC-2000, Intel IPSC/860, Intel Touchstone Delta, Alliant Campus, and Thinking Machines' CM-5. It will soon be ported to to the Intel Paragon. It is not difficult to port to new systems. Although p4 tries to be completely portable, there are a small number of specific exceptions (See 19 [Machine-Specific Notes], page 31) that may need to be taken into account on a given machine.

You can obtain the complete distribution of p4 by anonymous ftp from `info.mcs.anl.gov`. Take the file '`p4.tar.Z`' from the directory '`pub/p4`'. The distribution contains all source code, installation instructions, this reference manual, and a collection of examples in both C and Fortran. `Alog` is included in the distribution with p4. The file '`upshot.tar.Z`' contains display facilities that can be used with p4 and other systems.

To ask questions about p4, report bugs, contribute examples, etc., you can send mail to `p4@mcs.anl.gov`.

The current release is version 1.2. You can check which version of the source code you have by looking at the file '`lib/p4_patchlevel.h`' in the distribution. You can check which version of the object code you have linked to by running any p4 program with the command-line option `-p4version` (See 7 [Command-Line Arguments], page 12).

Salient features of the current release of p4 include:

- `xdr` support for heterogeneous networks
- manual enhanced and converted to latex format
- Emacs info version of the manual for on-line help
- SYSV IPC support added for several machines (for shared-memory multiprocessing on workstations that support multiple processors)
- instrumentation added for automatic logging/tracing
- automatic or user control of message-passing/buffer-management

- high-resolution clock support for several machines
- error/interrupt handling
- an optional secure server for quick startup

A useful companion system is the `alog/upshot` logging and X-based trace examination facility. (See 18 [Creating Logfiles for Upshot], page 28.)

2 Structure of the Distribution Directory

The p4 source code distribution contains the following files and subdirectories:

Makefile The makefile for making the p4 system, doing the installation, and making makefiles for user applications.

OPTIONS A file controlling various compile-time options, such as whether System V shared-memory operations are to be enabled, whether system debug message printing is to be enabled, and whether automatic instrumentation of p4 operations for the `upshot` logging and tracing program is to be done.

README General instructions.

alog Source code for the ALOG tracing package.

bin Scripts for starting and killing servers, killing runaway p4 processes, merging `upshot` logfiles, and other useful utilities.

contrib Examples contributed by p4 users.

contrib_f Fortran examples contributed by users.

doc The man page, together with this manual and supporting files.

include The include directory for making p4 applications. Most of these are (hard) links into the `lib` directory.

lib The source code for the p4 system.

lib_f The Fortran interface for p4.

messages A basic set of message-passing examples in C.

messages_f A basic set of message-passing examples in Fortran.

monitors A basic set of shared-memory examples in C.

servers The secure and insecure servers.

usc The portable microsecond clock routines.

util Assorted supporting files, particularly for making the p4 distribution.

3 Installing p4

In this section we describe how to install the p4 library, either for your own personal use or for the use of everyone at your site. In the first case you do not need any super-user privileges. In the second case, you may or may not, depending on how things are configured at your site. We also describe how to install and run the examples that come with p4, the online help system (this manual as an emacs info-file) and how to build a working directory for your own programs yet share the installed copy of p4 with other users.

3.1 Installing the p4 System

To build p4, position yourself in the `p4` directory and type:

```
make all MACHINE=<machine>
```

where <machine> is one of the machine names listed in '`p4/util/machines`', currently:

`SUN`	Sun-3, Sun386i, Sparc-1, or Sparc-2 workstations
`HP`	HP workstations
`DEC5000`	Dec 5000 workstations
`NEXT`	68030- or 68040-based NeXT workstations
`RS6000`	IBM RS 6000 series workstations
`IBM3090`	IBM 3090 running IBM's version of UNIX, AIX
`BALANCE`	Sequent Symmetry shared-memory multiprocessor
`SYMMETRY`	Sequent Symmetry shared-memory multiprocessor
`SYMMETRY_PTX`	Sequent Symmetry shared-memory multiprocessor PTX OS
`MULTIMAX`	Encore Multimax shared-memory multiprocessor
`GP_1000`	BBN GP-1000
`TC_2000`	BBN TC-2000
`TC_2000_TCMP`	BBN TC-2000 with the TCMP message-passing library
`IPSC860`	Intel IPSC/860 (nodes only)
`DELTA`	Intel DELTA
`TITAN`	Stardent Titan
`SGI`	Silicon Graphics workstations
`CRAY`	Cray X/MP
`FX8`	Alliant FX/8
`FX2800`	Alliant FX/2800 or FX/800
`FX2800_SWITCH`	Alliant FX/2800 or FX/800, with CAMPUS HiPPI switch
`KSR`	Kendall Square KSR-1
`CM5`	Thinking Machines' CM-5

For example:

```
make all MACHINE=SYMMETRY
```

The `all` is optional, for example

```
make MACHINE=SEQUENT
```

This will create a machine-dependent '`Makefile`' in each subdirectory, make the p4 library, and compile and link a subset of the examples.

To add a new machine type, or to change the characteristic parameters associated with an existing one, you can edit the file '`p4/util/defs.all`'.

To save disk space, various intermediate object files can be removed with

```
make clean
```

The system can be restored to its original, machine-independent state with

```
make realclean
```

Note that this removes the machine-dependent Makefiles in each directory, so the operation is not idempotent.

It is also possible to install (or clean) only some of the directories:

```
make all MACHINE=SUN DIRS=messages
make clean DIRS='monitors messages'
```

To install only the Makefiles in all subdirectories, use:

```
make makefiles MACHINE=<machine>
```

To install the necessary library and include files in a directory everything that is needed to compile and link p4 programs, do:

```
make install INSTALLDIR=<dir>
```

This will create a p4 directory in `<dir>`, build a minimal set of directories, copy the relevant '`.a`' and '`.h`' files into it, and test the installation by mking a small set of examples.

See 4 [Getting Started], page 5 for instructions on how to run some example programs after you have installed p4.

3.2 Installing the Documentation

The directory '`p4/doc`' contains this manual as well as files that require installation. This manual was prepared with the `latexinfo` package from GNU emacs; thus it can be made available online through `info`. The files in '`p4/doc`' are:

p4.tex the latex source for this manual, which uses the latexinfo style

latexinfo.sty the sytle file needed to latex this manual

p4.info the `info` version of this manual, ready to be put into the directory where `info` files are kept at your site. Check the value of your `emacs` variable *Info-directory*.

p4.txt plain ascii text of the manual, in case nothing else works.

p4.1 unix man page for the p4 library

p4f.1 unix man page for the Fortran interface to p4

The Postscript version of this manual is available by anonymous `ftp` from `info.mccs.anl.gov`, in the directory '`pub/p4`'. The file to get (in binary mode) is '`p4man.ps.Z`'.

3.3 Examples included with the Distribution

A good way to see how various p4 functions are used is to look at the example programs included in the distribution. The '`p4/monitors`' directory contains shared-memory examples written in C that use monitors, including one instrumented with ALOG. The '`p4/messages`' subdirectory contains message-passing examples written in C. The programs in '`p4/messages_f`' are Fortran message-passing examples, and the '`p4/contrib`' and '`p4/contrib_f`' directories contain a number of miscellaneous examples contributed by users. In each directory there is a '`README`' that describes the individual examples.

4 Getting Started

The easiest way to get started with p4 is to play with some of the sample programs provided with the system.

4.1 A Message-Passing Example

We will begin with a message-passing example in the sub-directory named '`p4/messages`'. The code for the program is in the files '`sr_test.c`' and '`sr_user.h`'.

4.2 Program Description

As the name implies, this program is an example of p4's send/receive functionality. Briefly, it is a simple program that runs a master process and some slave processes. The master and the set of slaves form a ring of processes in which the master reads a message from stdin and sends a copy of the message to the first slave, which passes it on; the last slave passes the message back to the master. If the master receives an undamaged copy of the message, it assumes that all went well, and reads another message. Note that the ring of processes is a logical structure in which each process assumes that its predecessor in the ring is the process with the next lower id, and its successor is the process with the next higher id. The master has id 0 (zero) and has the process with the largest id as its predecessor.

4.3 Analysis of the Program

The first executable p4 statement in a program should be:

```
p4_initenv(&argc,argv);
```

This initializes the p4 system and allows p4 to extract any command line arguments passed to it, e.g. debugging parameters.

Similarly, the last executable p4 statement in a program should be:

```
p4_wait_for_end();
```

This waits for termination of p4 processes and performs some cleanup operations.

The procedure `p4_get_my_id` returns the unique integer id assigned to the calling process by p4.

The statement:

```
p4_create_procgroup();
```

reads a procgroup file that the user builds and creates the set of slaves described in that file. Obviously this statement must be executed before any slaves can be assumed to exist. This procedure is the method you must use to create processes that do message-passing.

The procedure `p4_clock` returns an integer that represents wall-clock time in milliseconds. It is typically used to retrieve the time before and after some work, the difference representing the time to do that work. Note that there is also a `p4_ustimer` that is useful on those machines that support a microsecond timer.

The procedures `p4_send` and `p4_sendr` are two of several p4 procedures that are available for sending messages to other processes. They take as arguments the message type, the id of the "to" process, the address of the message, and the message length.

The procedure `p4_recv` receives a message from another process and sets the values of all four parameters. `P4_recv` will automatically retrieve a buffer in which to place a received message, thus `p4_msg_free` may be called to free that buffer when it is no longer needed.

The procedure `p4_num_total_slaves` is one of several procedures that the user can invoke to determine information about the current execution.

To run this program, you need to create a procgroup file that describes where all slave processes are to be executed (See 5 [Specifying Processes in the Procgroup File], page 6). We will assume that you have an example procgroup file (named '`procgroup`') in the '`p4/messages`' directory, and can run `sr_test` by merely typing:

```
sr_test
```

If the procgroup file is elsewhere, then you must type:

`sr_test -pg` *pathname_of_procgroup_file*

Another example that is made by default is the program `systest`. It tests a number of the message-passing features of p4.

5 Specifying Processes in the Procgroup File

The procgroup file is the only portion of the interface that is very likely to change through multiple versions of p4. As new architectures are supported, it is hoped that we can merely alter the procgroup file format to reflect any new features. (Of course new procedure calls may also be required, but existing procedure calls will remain unchanged when possible).

The current format of a procgroup file is as follows:

```
local n [full_path_name] [loginname]
remote_machine n full_path_name [loginname]
.
.
.
```

On cube and mesh architectures, the program is started via some special command executed from the host machine. In such cases, the procgroup file name can be specified to the special command line along with the program name (see for example the `runcube` and `rundelta` shell scripts in the '`p4/messages`' subdirectory). In those cases where no special command is required, no special handling is required for the procgroup filename.

The first line of a procgroup file must be "local n" where n is the number of slave processes that are in the same cluster as the master. The full path name on the "local" line is ignored on machines other than cube and mesh machines. The subsequent lines contain either three or four fields:

1. the name of a remote machine on which slave processes are to be created.
2. the number of slaves that are to be created on that machine, i.e. be in the same cluster (note that on machines that support it, the processes in a cluster will share memory)
3. the full path name of the executable slave program
4. optionally, the user login name on the remote machine, if different from that on the host machine.

As an example, let's assume that you have a network of three Sun workstations named sun1, sun2, and sun3. We will also assume that you are working on sun1 and plan to run a master process there. If you would like to run one process on each of the other Suns, then you might code a procgroup file that looks like:

```
# start one slave on each of sun2 and sun3
local 0
sun2  1  /home/mylogin/p4pgms/sr_test
sun3  1  /home/mylogin/p4pgms/sr_test
```

Lines beginning with `#` are comments.

Next, let's assume that you have a Sequent Symmetry (named symm) and an Encore Multimax (named mmax). We will also assume that you are working on symm, and plan to run the master there. If you would like to run two processes on symm (in addition to the master) and two on mmax, then you might code a procgroup file that looks like:

```
local 2
mmax  2  /mmaxfs/mylogin/p4pgms/sr_test
```

P4 also permits you to treat the symmetry as a remote machine even when you are running the master there. Thus, you might code a procgroup file as follows:

```
local 2
symm  2  /symmfs/mylogin/p4pgms/sr_test
mmax  2  /mmaxfs/mylogin/p4pgms/sr_test
```

In this example, there are seven processes running. Five of the processes are on symm, including the master. Two of the processes on symm are in the master's procgroup and two are running in a separate procgroup as if they were on a separate machine. Of course, the last two are running on mmax.

Some notes about the contents of the procgroup file should be made at this point. First, the value of `n` on the local line can be zero, i.e. the master may have no local slaves. Second, the local machine may be treated as if it is a remote machine by merely entering it in some line as a remote machine. Third, a single machine may be treated as multiple remote machines by having the same remote machine name entered on multiple lines in the procgroup file. Fourth, if a single machine is listed multiple times, those processes specified on each line form a single cluster (share memory). Fifth, the cluster size specified for a uniprocessor should be 1, because all slaves in a cluster are assumed to run in parallel and to share memory.

We refer to the original (master) process as the "big master". The first process created in each cluster is the "remote master" or the "cluster master" for that cluster. All p4-managed processes (see the procedure `p4_create_procgroup`) have unique integer id's beginning with 0. The processes within a cluster are numbered consecutively.

6 Developing a Simple p4 Program

The real fun associated with any computing environment arrives when you actually type in a program and run it yourself. We will assume that you have successfully installed p4 on your own system and are ready to write a small program, compile it, and run it.

6.1 A Minimal Example

We will start with a tiny program in which the slave processes do no work, and then expand its capabilities. Edit a file called '`p4simple.c`' and type:

```
#include "p4.h"

main(argc,argv)
int argc;
char **argv;
{
    p4_initenv(&argc,argv);
    if (p4_get_my_id() == 0)
        p4_create_procgroup();
    slave();
    p4_wait_for_end();
}

slave()
{
    printf("Hello from %d \n",p4_get_my_id());
}
```

This is one of the simplest p4 programs that you can write. Let's examine it. The `#include "p4.h"` statement must appear in all programs that use any p4 features. The procedure `p4_initenv` must be invoked before any other p4 procedures, and `p4_wait_for_end` must be invoked after all p4 processing is completed. The `p4_get_my_id` returns a unique integer id for each process, beginning with 0. The procedure `p4_create_procgroup` should only be invoked once (by process 0), and is responsible for creating all other processes. The way in which `p4_create_procgroup` determines how many other processes there should be, and where they should run, will be discussed shortly.

All processes that this program executes invoke the slave procedure, including process 0. Thus, in this program, the master process acts just like all other processes once it gets the environment established.

To understand how things get started, let's consider two separate situations. In the first situation, all processes are running on a single machine. Then, when process 0 starts, it executes the `p4_create_procgroup` procedure to start all other slaves. The other slaves are started on the same machine by means of a UNIX *fork*, and they immediately invoke a procedure named slave. Thus, these local slaves do not ever execute the main procdure.

In the second situation, there may be slaves running both on the same machine as process 0, and slaves running on other machines as well. In this situation, the first slave running on a remote machine will need to execute the main procedure. It will discover that it is not process 0. However, as part of initialization, process 0 will direct it to fork any additional slaves required on the same machine.

In some ways, the above example can be used a a prototype for all p4 programs, just by varying the content of the `slave` routine. The routine name `slave` is built into p4 in order to enable the same code to be run on shared-memory machines via `fork` or on remote machines via `rsh`. Different wrappers are automatically put around the `slave` code. These wrappers depend on the user code being called `slave`. Thus your p4 program must contain a subroutine called `slave` in order to link properly.

6.2 A More Complicated Example

Now, let's make the slave process a little bit more interesting. Let's assume that we have `nprocs` slaves with ids 0, 1, 2, ... `nprocs` -1. And, we want to write a program in which every process sends a single message to every other slave, and then receives a message from every other slave. We might alter the code for the slave procedure to be the following:

```
slave()
{
    char *incoming, *msg = "hello";
```

```
        int myid, size, nprocs, from, i, type;

        myid = p4_get_my_id();
        nprocs = p4_num_total_ids();
        for (i=0; i < nprocs; i++)
        {
            if (i != myid)
                p4_send(100, i, msg, strlen(msg)+1);
        }
        for (i=0; i < nprocs - 1; i++)
        {
            type = -1;
            from = -1;
            incoming = NULL;
            p4_recv(&type,&from,&incoming,&size);
            printf("%d received msg=:%s: from %d \n",myid,incoming,from);
            p4_msg_free(incoming);
        }
    }
```

This program demonstrates several features of p4's support for message-passing. Before we get into the specifics however, let's examine the overall logic of the program. Each process determines its own id and the total number of processes executing in this run (including process 0). Then, in the first for-loop, each process sends a single message to each of the other processes. Finally, in the second for-loop, each process receives a message from each of the other processes.

The `p4_send` call requires 4 arguments:

- a message type (arbitrarily chosen to be 100 here)
- the id of the process to receive the message
- the message itself
- the size of the message

The use of `p4_recv` is slightly more complicated. First, we assign -1 to each of the parameters type and from. This is done because -1 represents a wildcard value indicating we are willing to receive a message of any type from any process. Here, we could have coded type to be 100, and specified from equal to the value of i each time through the loop (skipping our own id). By setting incoming to NULL, we have also indicated to `p4_recv` that we do not have a buffer in which to place the received message, so `p4_recv` should obtain a buffer for us and place the message in that buffer. `p4_recv` treats these three parameters as both input and output values. Thus, it alters the value of each such that type and from indicate the type of message received and the id of the process that sent it. The value of `incoming` is altered to point to the buffer where the message was placed. The `size parameter` is strictly an output parameter and iindicates the size of the received message. It is possible for the user to provide his own buffer; this will be demonstrated later.

Finally, note that `p4_msg_free` frees the message buffer obtained by `p4_recv`. The procedure `p4_msg_free` should be called only after the contents of the message are no longer needed. `P4_msg_free` should be used to free these buffers because, although a user only sees the data portion of a message, p4 internally represents a message as a structured data item.

To compile and link this program for execution, you need to create a makefile. We will assume that you have installed p4 in '`/usr/local/p4`' and that you have typed the program above into a file name '`p4simple.c`' in the directory '`/home/mylogin/p4pgms`'.

To build your makefile, copy the file

```
/usr/local/p4/messages/makefile.proto
```

into your working directory. This is a prototype makefile that contains machine-independent information, and which p4 can use to build a machine-specific makefile for your program. This prototype makefile contains information about several sample programs that demonstrate message-passing in p4. If you edit this file, you will see information for making a program named `sr_test`. Do a global change of `sr_test` to `p4simple`. You should also change the value of `P4_HOME_DIR`. It should contain the full pathname of the p4 system, e.g. '`/usr/local/p4`'. Now change directories to '`/usr/local/p4`' and type:

```
make makefiles MACHINE=<machine_type> DIRS=/home/mylogin/p4pgms
```

where `<your_machine_type>` is the machine type that you specified when you installed p4 on your machine. Now, you should be able to change back to your directory and see a file named Makefile there. You should then be able to type:

```
make p4simple
```

There is one last piece missing before you can execute your program. Recall that `p4_create_procgroup` needs to know how many processes to start and where to start them; it reads a file (called a *procgroup file*) to gather this information. p4 always assumes that you have a master process, and that you describe the slave processes (process groups) in the procgroup file. You can name a procgroup file any name you choose, but `procgroup` is the default name. The information contained in procgroup files can get fairly involved, but if you have a computer that supports shared memory among processes, then you can code a very simple example at first.

Let us suppose first that you want to run your program on a network of workstations. Then your procgroup should look something like:

```
local 0
some.network.machine 1 /home/me/p4progs/p4simple
```

This file indicates that you wish to run only the master on the local machine (the one you are logged into when you execute the program) and one slave on the machine `some.network.machine`.

Now, all you have to do to run your program is type:

```
p4simple
```

You should see a line printed each time a process receives a message from another process (on some machines, there may be a restriction that only one process can do I/O, however such restrictions are not common). Experiment by changing the number of slaves indicated in the procgroup file.

You may notice that even a small p4 program becomes large when linked with the p4 library. You might consider using `strip` to reduce the size or removing `-g` from the CFLAGS in the makefile.

7 Command-Line Arguments

The command-line arguments to a p4 program are all optional.

```
-help           get this information and the version number, then exit
-pg             procgroup_file
-dbg            set debug level
-rdbg           set remote debug level
-gm             set globmemsize
-dmn            provide local domainname
-outfile        output file for master
-routfile       output file prefix for remote masters
-ssport         port# private port number for secure server
-p4log          enable internal p4 logging by alog
-p4version      print the current p4 version number
```

8 The p4 Function Library

8.1 Overview of the Library

In the following sections, we provide details for each p4 function in the library. The procedures are gathered into the following groups:

- Functions for managing processes and clusters
- Functions for message passing
- Functions for shared memory
- Functions for timing p4 programs
- Functions for debugging p4 programs
- Miscellaneous functions
- Fortran interface functions

8.2 Return Codes from p4 Functions

Most p4 functions return -1 if an error occurs. Some, however, call the function `p4_error` when severe errors occur. This function prints a message and then attempts to terminate all of the user's processes See 13 [Functions for Debugging p4 Programs], page 23.

9 p4 Functions for Managing Processes and Clusters

In some situations a p4 procedure will give an error message and then exit. This is typically done as a result of a failed system call and handled by calling the p4 procedure named `p4_error` that examines the return values from socket procedures, etc. Most of the time however, the procedures simply return a value. Some of the procedures return no value and thus are declared to return `VOID`. Some of the procedures return either a pointer to a character string or `NULL`; `NULL` indicates an error. The remaining procedures return an integer value; (-1) indicates an error.

9.1 Functions for Process Management

In this section we describe the p4 functions needed for basic creation and termination of processes.

```
int p4_initenv(argc,argv)
int *argc;
char **argv;
```

should be called by your program before an attempt is made to use any p4 procedures or data areas. We suggest making it the first executable statement in your program. `p4_initenv` parses the command line arguments and extracts the ones intended for p4 ignoring all others (see the discussion of command line arguments). Note that you pass the address of `argc` to `p4_initenv` so that it can actually remove its own arguments before your program looks at them.

```
int p4_create(fxn)
int (*fxn)();
```

```
int p4_create_procgroup()
```

There are two procedures that you can use to create processes in p4, `p4_create_procgroup` and `p4_create`. Processes created via `p4_create` are said to be "user-managed" whereas those created by `p4_create_procgroup` are "p4-managed". The p4-managed processes are automatically assigned unique id's (beginning with 0 for the big master), they have message queues allocated for them so that they can do message-passing, and they are able to run either on a shared-memory multiprocessor with the creating process or they can run on a separate machine. Processes created via `p4_create` do not have any of these advantages. They must develop their own id's, they cannot do message-passing, and they can only run on a shared-memory multiprocessor with the creating process. The only disadvantage of `p4_create_procgroup` is that you must build a '`procgroup`' file describing the set of required slave processes before the master program begins execution. This eliminates the possibility of determining late in the execution exactly how many processes you want to use to solve a problem. Generally, this is not a problem, especially since we can combine `p4_create_procgroup` and `p4_create` in the following way: You can use `p4_create_procgroup` to develop a network of processes that talk to each other via messages. Each of those processes can create further processes to help it out as necessary. The original set of processes communicate with their local slaves through shared data areas and with each other via message-passing.

`p4_create` receives one argument that is a pointer to a function. It creates a single new process that executes the indicated function. The new process may share data areas (in shared memory) with the parent process. However, the new process is not managed by the p4 system in the sense that it is not assigned an id, it cannot pass messages, etc. The only p4 procedure that deals with user-managed slaves is `p4_create`. No other procedures are even aware of their existence.

`p4_create_procgroup` reads your '`procgroup`' file to determine the number of slave processes to create and where they are to be placed. It builds a procgroup table that describes all created processes and gives a copy of the table to each process. The processes then use the table to discover how to communicate with each other (processes in a cluster can send messages directly through shared memory or some other vendor-specific mechanism), others communicate via sockets). An alternative method is to build the table in memory yourself and use `p4_startup`.

The effect of `p4_create_procgroup` can be obtained in another way if a system would prefer to use its own way of specifying the locations of processes. A user may allocate the procgroup data structure and then fill it in "by hand" rather than by reading a file in p4 procgroup format. The following procedures support this method of starting processes.

```
struct p4_procgroup *p4_alloc_procgroup()
```

allocates a procgroup data structure of the form described in `p4.h`. The formats of individual entries (`p4_procgroup_entry`) are given there as well.

```
int 4_startup(pg)
struct p4_procgroup *pg;
```

starts processes as specified by an an already-created procgroup data structure allocated by `p4_alloc_procgroup` and filled in by the user using the structures `p4_procgroup_entry` and `p4_procgroup`.

```
VOID p4_wait_for_end()
```

is the p4 termination/cleanup procedure that you should invoke at the end of every execution of a program that uses p4. It does some termination processing and then waits for slave processes to end.

```
int p4_get_my_id()
```

returns an integer value representing the id of the process assigned by the p4 system. If the process is not a p4-managed process, the value (-1) is returned.

```
int p4_num_total_ids()
```

returns an integer value indicating the total number of ids started by p4 in all clusters, including the big master and all remote masters.

```
int p4_num_total_slaves()
```

returns an integer value indicating the total number of processes started by p4 in all clusters, including all remote masters but not the big master.

9.2 Functions for Cluster Management

The p4 system supports the *cluster* model of parallel computation, in which subsets of processes share memory with one another, with the clusters communicating via messages. A procgroup file for a program written for the cluster model might look like this:

```
local 4
alliant1.abc.edu     5 /home/me/myprog
alliant2.abc.edu     5 /home/me/myprog
encore.somewhere.edu 5 /usrs/me/myprog
```

This would specify a total of 20 processes, 5 (including the master) running on the local machine (here assumed to be capable of supporting five processes that share memory) together with 5 slaves each on three other shared-memory machines.

```
VOID p4_get_cluster_ids(start,end)
int *start;
int *end;
```

receives pointers to two integers. It places the p4-assigned id's of the first and last ids within the current cluster into the two arguments (including the remote master).

```
int p4_get_my_cluster_id()
```

returns a unique id (relative to 0) within a cluster of p4-managed processes. Thus, a cluster master will always have a cluster id of 0. It is not clear that a separate cluster id is really useful, but the functionality is provided just in case.

```
BOOL p4_am_i_cluster_master()
```

returns a BOOL value indicating whether the invoking process is the "cluster master" process within its cluster.

```
int p4_num_cluster_ids()
```

returns an integer value indicating the number of ids in the current cluster as started by `p4_create_procgroup`.

10 Functions for Message Passing

P4 supports a set of send/receive procedures. These procedures are "generic" in the sense that they do not know whether a message must travel across a network or through shared memory, or via some other mechanism. They depend on a lower-level set of procedures that handle local or network (remote) communications. By default, the messages are assumed to be typed. If the user wishes to use untyped messages, he can hide the typing by coding some very simple C macros that always use a single message type.

10.1 Explicit Sending and Receiving of Messages

```
p4_send(type,to,msg,len)
p4_sendr(type,to,msg,len)
p4_sendx(type,to,msg,len,datatype)
p4_sendrx(type,to,msg,len,datatype)
p4_sendb(type,to,msg,len)
p4_sendbr(type,to,msg,len)
p4_sendbx(type,to,msg,len,datatype)
p4_sendbrx(type,to,msg,len,datatype)

int type, to, len, datatype;
char *msg;
```

Each of these procedures sends a message. The `type` argument is an integer value chosen by the user to represent a message type. The `to` argument is an integer value that specifies the p4-id of the process that should receive the message. The `len` argument contains the length of the message to be passed. Note that some of the procedures have a "b" in their name, e.g. `p4_sendb`. These procedures assume that the msg is in a buffer that the user obtained earlier via a `p4_msg_alloc`; otherwise, the buffer is assumed to be in the user's local space, and may cause the message to be copied internally. The procedures with an "r" in the name do not return until an acknowledgement is received from the `to` process (the "r" stands for rendezvous). Those procedures with an "x" in the name take an extra argument (datatype) that specifies the type of data in the message; these procedures will use that information to call XDR for data conversion if the message is being passed to a machine of a different architecture, i.e. where the internal representation may be different.

```
BOOL p4_messages_available(req_type,req_from)
int *req_type,*req_from;
```

returns a BOOL value indicating whether the process has any messages available or not. The parameters `req_type` and `req_from` are both pointers to integers; they are used as *both* input and arguments. On input, `req_type` has a value that indicates the type of message that the user wishes to check for availability (-1 indicates any type). The variable `req_from` is used similarly to indicate who a message is desired from.

```
int p4_recv(req_type,req_from,msg,len_rcvd)
int *req_type,*req_from,*len_rcvd;
char **msg;
```

takes four arguments. The `msg` argument is a pointer to a pointer to a `char`. If this value is NULL, then p4 will allocate the buffer for the message according to its length. That is, one need not know ahead of time the length of a message being received. If this value is not NULL, then it points to a p4 message buffer that the user has obtained via `p4_msg_alloc`.

The `len_rcvd` argument is a pointer to an integer that is assigned the length of the received message. `Req_type` and `req_from` are both pointers to integers; they are used as both input and arguments. On input, `req_type` has a value that indicates the type of message that the user wishes to receive (-1 indicates any type). It will block until a message of that type is available. `Req_from` is used similarly to indicate who a message is desired from. One important note about this procedure is that it obtains the area in which to place a message, and the user must explicitly free that area when finished with it (see `p4_msg_free`). There is an option available with `p4_recv` in which the user can provide his own buffer rather than having p4 allocate it. To do this, the user points `msg` to a buffer that he must obtain via a call to `p4_msg_alloc` (see below). Then he assigns `len_rcvd` a value which is the length of that buffer. In this case, `len_rcvd` is both an input and output variable. In addition, no `p4_msg_free` need be performed if the same buffer is going to be re-used multiple times.

```
char *p4_msg_alloc(len)
int len;
```

obtains a pointer to a buffer area that can be used to receive a message. This procedure should be used for this task because a message has hidden information which the user is unaware of and therefore should not use malloc to obatin the area.

```
VOID p4_msg_free(m)
char *m;
```

frees the message pointed to by `m`. This procedure should be used for this task because a message has hidden information which the user is unaware of and therefore cannot be freed by the user.

10.2 Global Operations

P4 supports a number of operations for dealing with all processes at once.

```
p4_broadcastx(type, data, data_len, data_type)
int type;
char *data;
int data_len, data_type;

p4_broadcast(type, data, data_len)
int type;
char *data;
int data_len;
```

provide the ability to broadcast messages like `p4_send` and `p4_sendx`.

```
p4_global_op(type,x,nelem,size,op,data_type)
int type;
char *x;
int size, nelem;
int (*op)();
int data_type;
```

where `op` is one of:

```
p4_int_absmax_op()
p4_int_absmin_op()
p4_int_max_op()
p4_int_min_op()
p4_int_mult_op()
p4_int_sum_op()
p4_dbl_absmax_op()
p4_dbl_absmin_op()
p4_dbl_max_op()
p4_dbl_min_op()
p4_dbl_mult_op()
p4_dbl_sum_op()
p4_flt_absmax_op()
p4_flt_absmin_op()
p4_flt_max_op()
p4_flt_min_op()
p4_flt_mult_op()
p4_flt_sum_op()
```

and `data_type` is one of `P4INT`, `P4LNG`, `P4FLT`, or `P4DBL`.

This collection of routines provide the ability to do a variety of global operations. See the example programs in subdirectory '`p4/contrib`'. They apply the commutative operation `op` globally to `x` on an element-by-element basis and broadcast the result to all nodes. That is each process ends up with

```
for (i=0; i<n; i++)
     x[i] = x[node 0][i] op x[node 1][i] op x[node 2][i] op ...
```

`op` should be of the form

```
VOID op(char *x, char *y, int nelem)
{
    data_type *a = (data_type *) x;
    data_type *b = (data_type *) y;

    while (nelem--)
        *a++ operation= *b++;
}
```

where `data_type` and `operation` are chosen appropriately.

The order in which nodes apply the operation is undefined (hence `op` must be commutative). The communication may be internally sub-blocked so the function `op` should not be hardwired to specific vector lengths.

This is still a relatively primitive version, which gathers the necessary data up a balanced binary tree and then uses `p4_broadcast` to send the results back.

```
VOID p4_global_barrier(type)
int type;
```

This procedure takes one argument which is the message type to be used for internal message-passing. It causes the invoking process to hang until all processes specified in the procgroup file have invoked the procedure.

11 Functions for Shared Memory

11.1 Managing Shared and Local Memory

```
char *p4_malloc(n)
int n;
```

typically acts like the standard `malloc`, but may be rewritten for user systems that require different operation.

```
VOID p4_free(p)
char *p;
```

typically acts like the standard `free`, but may be rewritten for user systems that require different operation.

```
char *p4_shmalloc()
```

acts like the standard `malloc` except will obtain shared memory on machines that support sharing memory among processes. Compare with `p4_malloc`.

```
VOID p4_shfree()
```

frees memory obtained with `p4_shmalloc`. Compare with `p4_free`.

11.2 Shared Memory Data Types

The abstraction provided by p4 for managing data in shared memory is *monitors*. Good places to learn about the monitor concept in general are [2] and [4]. The specific approach taken by p4 is described in [1]. P4 provides several useful monitors (`p4_barrier_t`, `p4_getsub_monitor_t`, `p4_askfor_monitor_t`) as well as a general monitor type to help the user in constructing his own monitors (`p4_monitor_t`).

11.3 Monitor-Building Primitives

The following functions can be used to construct monitors. A monitor so constructed has the type `p4_monitor_t`.

```
int p4_moninit(m,i)
p4_monitor_t *m;
int i;
```

initializes the monitor pointed to by `m` and gives it `i` queues for processes to wait on while they are blocked (see `delay`). One queue is sufficient for most purposes. The queues are numbered beginning with 0.

```
VOID p4_menter(m)
p4_monitor_t *m;
```

enter the monitor pointed to by m. By the definition of a monitor, access is restricted to a single process in the monitor at a time (if everybody plays by the rules).

```
VOID p4_mexit(m)
p4_monitor_t *m;
```

exits the monitor pointed to by m. You are of course assumed to have previously entered that monitor.

```
VOID p4_mcontinue(m,i)
p4_monitor_t *m;
int i;
```

checks to see if there are any processes blocked on the `i`-th queue of the monitor `m` and causes one of them to be released for entry to the monitor if so. If there are no such processes, the invoking process simply exits. Note that a process could have been blocked previously by invoking the procedure `p4_mdelay`. The queues are numbered beginning with 0.

```
VOID p4_mdelay(m,i)
p4_monitor_t *m;
int i;
```

permits a process to delay itself on the `i`-th queue of monitor `m` if the process wishes to release the monitor, but wants to be waked up by another process later (via the procedure `p4_mcontinue`). The queues are numbered beginning with 0.

11.4 Some Useful Monitors

In this section we describe some of the specific monitors that are built into the p4 library. Each of them has its own pre-defined type, which can be used to allocate storage for them, which should be in shared memory. See the '`p4/monitors`' directory for examples. A lock is itself a monitor, with no extra delay queues.

```
VOID p4_lock_init(l)
p4_lock_t *l;
```

initializes the lock `l`. Must be used prior to any attempts to lock or unlock `l`.

```
VOID p4_lock(l)
p4_lock_t *l;
```

blocks if the lock `l` is already locked, otherwise locks `l` and proceeds.

```
VOID p4_unlock(l)
p4_lock_t *l;
```

unlocks the lock `l`.

```
VOID p4_getsub(gs,s,max,nprocs)
p4_getsub_monitor_t *gs;
```

```
int *s,max,nprocs;
```

is a procedure used to obtain the next value of a shared counter (subscript). It takes as its first argument, a pointer to a getsub monitor that protects the shared counter. It assigns the current value of the counter to the integer that s points to, and then increments the counter by 1. `p4_getsub_init` initially sets the counter to 0. When the counter passes the value `max`, all `nprocs` processes are returned the value (-1) once, then the counter is reset to 0 for further use.

```
VOID p4_getsubs(gs,s,max,nprocs,stride)
p4_getsub_monitor_t *gs;
int *s,max,nprocs,stride;
```

is like `p4_getsub` except that the counter is increased on each call by `stride` instead of 1.

```
int p4_getsub_init(gs)
p4_getsub_monitor_t *gs;
```

initializes the getsub monitor pointed to by `gs`; this initialization includes assigning a value of 0 to the counter that the monitor protects.

The standard barrier synchronization pattern is expressed as a monitor. There can be multiple barrier monitors, and one can wait for only some of the processes at the barrier if this is desired.

```
VOID p4_barrier(b,nprocs)
p4_barrier_monitor_t *b;
int nprocs;
```

causes the executing process to hang until `nprocs` processes execute a barrier instruction with a pointer to the same barrier monitor `b` as an argument.

```
int p4_barrier_init(b)
p4_barrier_monitor_t *b;
```

initializes the barrier monitor `b`; this procedure should be invoked before you attempt to use the monitor in any operations.

Finally, the `askfor` monitor functions like a general dispatcher of work.

```
int p4_askfor(af,nprocs,getprob_fxn,problem,reset_fxn)
p4_askfor_monitor_t *af;
int nprocs;
int (*getprob_fxn)();
VOID *problem;
int (*reset_fxn)();
```

requests a new "problem" to work on from the problem pool. The arguments are (1) a pointer to the askfor monitor that protects the problem pool, (2) the number of processes that call this procedure (with `af`) looking for work, (3) a pointer to the user-written procedure that obtains a problem from the pool, (4) a pointer that is filled in with the address of a user-defined representation of a problem to be solved, and (5) a pointer to a user-written procedure to reset when all problems in the pool are solved, in case the same monitor is re-used for another set of problems later. `p4_askfor` returns an integer indicating whether a problem was successfully obtained or not:

```
    -1     : program is terminating (some process called p4_progend)
     0     : a problem was obtained and ''problem'' points to it
     1     : problem solved by exhaustion, i.e. no more problems to get
     n > 1 : a process found a solution and called p4_probend with code
n
```

For a detailed discussion of the "askfor" monitor, see [1].

```
int p4_update(af,putprob_fxn,problem)
p4_askfor_monitor_t *af;
int (*putprob_fxn)();
VOID *problem;
```

updates the problem pool being managed by the askfor monitor. The arguments are (1) a pointer to the askfor monitor that protects the problem pool, (2) a pointer to the user-written procedure that puts problems into the pool, and (3) a pointer to a user-defined representation of a problem to be put in the pool. `Putprob_fxn` should return 1 if it did indeed put a new problem into the pool, so that any delayed processes should wake up and re-examine the pool (this logic is handled by the `p4_askfor`) and 0 if upon entering the monitor and examining its potential problem together with the data there it decided not to add a new problem to the pool. It can be assumed that the "putprob" logic (defined by `putprob_fxn`) is executed inside the monitor.

```
int p4_askfor_init(af)
p4_askfor_monitor_t *af;
```

initializes the askfor monitor `af`; this procedure should be invoked before you attempt to use the monitor in any operations.

```
VOID p4_probend(af,code)
p4_askfor_monitor_t *af;
int code;
```

allows the user process to mark a problem as solved early when several processes are co-ordinating their activities via an askfor monitor. The code is an integer value that will be returned to all processes when they "askfor" a new sub-problem to work on.

```
VOID p4_progend(af)
p4_askfor_monitor_t *af;
```

allows a process to cause a return code of (-1) to be returned to all processes using an askfor monitor. This would typically be called by a master process to indicate that no more problems are to be solved and that all slave processes should terminate.

12 Functions for Timing p4 Programs

A small number of simple functions are available for accessing various clocks and timers.

```
int p4_clock()
```

returns a value in milliseconds. This is a wall-clock value, usually obtained from the system via `gettimeofday`. Also see `p4_ustimer` below.

```
p4_usc_time_t p4_ustimer()
```

returns a wall-clock time value in microseconds. The precision of this number depends on the timer installed on the individual machine. In some cases the resolution may be no greater than that of `p4_clock()`. For arithmetic and printing purposes, the type `p4_usc_time_t` is an unsigned long integer.

```
p4_usc_time_t p4_usrollover()
```

returns the timer value at which a microsecond timer "rolls over". Since `p4_usc_time_t` is a long integer's worth of microseconds, it is likely that the timer will roll over (become zero) during even medium-length runs.

13 Functions for Debugging p4 Programs

P4 has a set of routines to aid in producing a printed trace of events, both user-defined and pre-defined in the p4 system.

```
VOID p4_dprintf(fmt, va_alist)
char *fmt;
va_dcl
```

acts just like the standard `printf` except that the print line is preceded by a value that identifies the process. This value is typically the string `pn_u` where `n` represents the p4-assigned id and `u` represents the unix-id of the process on its host. However, there are other forms of this value. For example, the big master is represented as `bm_u`. Also, if a process prints before it has a p4-assigned id, then its value will be something like `bm_slave_n_u` or `rm_slave_n_u`. Typically, it is not possible for a user program to print anything before being assigned an id by p4, but the p4 system itself may use this procedure to print messages from a particular process if it encounters problems getting the process initialized.

```
VOID p4_dprintfl(level, fmt, va_alist)
int level;
char *fmt;
va_dcl
```

is like `p4_dprintf` except that the first argument is an integer indicating the debugging level that must be in effect before this message will print. A level of 0 will cause the message to always print. If you run a program with the debug level set to 5 (via command-line arguments), then all `dprintfl`'s with level less than or equal to that debug level will print. See 7 [Command-Line Arguments], page 12 for how to set the debug level at run time.

The debug level can be examined and changed by the user during execution:

```
int p4_get_dbg_level()
```

returns the current debug level for this process and its cluster.

```
VOID p4_set_dbg_level(level)
```

```
int level;
```

sets the current debug level for this process and its cluster. P4 itself is liberally instrumented with `p4_dprintfl`'s of level 10 and above, leaving levels 0-9 for the user. The greater the debug level of the built-in messages, the greater understanding of p4 needed by the user to make sense of them. However, levels as high as 30 may well be useful to the user trying to debug a p4 program. Roughly speaking, the following debug levels produce messages about the indicated events.

```
level 10:  created process
           sent message
           received message

level 20:  creating process
           sending message
           receiving message
           process starting
           process exiting

level 30:  waiting for ack
           sending ack
           sent ack
           received ack
           queueing message for later receipt
           queued message for later receipt

level 40:  memory management
           buffer management

level 50:  reading procgroup
           other initialization message exchange

level 60:  send-receive details, especially machine-specific traces

level 70:  listener interactions:
            creating listener
            created listener
            messages from inside listener

level 80:  detailed data structures after initialization

level 90:  detailed tracing of flow thru procedures
```

For optimum performance, the test of the debug level required by these messages can be removed at compile time by not commenting out the `#define P4_DPRINTFL` line in the '`OPTIONS`' file. (See 2 [Structure of the Distribution Directory], page 2).

The following function is provided to deal with abnormal termination. It can be called by any process.

```
VOID p4_error(string, value)
char *string;
int value;
```

prints `string` as an error message and then forcefully terminates all co-operating processes and cleans up all shared resources.

```
VOID p4_soft_errors(onoff)
int onoff;
```

enables/disables soft errors, returning the previous setting. The default is "disabled", which means that certain p4 functions will call `p4_error` instead of returning -1.

`p4_error` gets control on certain kinds of interrupts. It is automatically called for `SIGSEGV`, `SIGBUS`, and `SIGFPE` interrupts, to catch user programming errors and clean up, after which it returns interrupt handling to default mode and returns, so that the user may obtain a dump. It also handles `SIGINT` interrupts, in which case it cleans up and exits. Finally, it may be called directly by the user, in which case it cleans up (other p4 processes and IPC's) and exits.

Although `p4_error` is supposed to get rid of all running p4 processes, it can happen that an error is bad enough that p4 processes are left running. A primitive aid in finding and killing these processes is the shell script `kj`, which takes a string as an argument and then kills processes containing that string as part of their program names. Currently it only kills processes on the machine where it is run, but it can be run via `rsh` on remote machines. There are other useful scripts (e.g. `killipc` and `killp4`) in the '`p4/bin`' directory to do such things as clean up SYSV IPC items that may be left when a program abnormally terminates. P4 will generally cleanup these items if the abnormal termination is a type that p4 traps, otherwise the user must do the cleanup. This is an unfortunate side-effect of the way that SYSV handles things, it really should be the OS's function to take care of this.

On many machines it is possible to attach a debugger like `dbx` to a running process. This is one way to find out where a hanging process is stuck.

14 Miscellaneous Functions

In this section are found functions that do seem to fit neatly into any of the other sections.

```
char *p4_version()
```

returns a string containing the version number of p4 being run.

```
VOID p4_get_cluster_masters(numids, ids)
int *numids, ids[];
```

This procedure fills in the values of numids and ids. It obtains the p4-ids of all "cluster masters" for the program, placing them in the ids array and placing the number of ids in numids.

```
VOID p4_print_avail_buffs()
```

P4 maintains an array of buffer lists of various sizes, so that it can very rapidly allocate and deallocate buffers. You can see the contents of the buffer pools at any time by calling this procedure.

```
VOID p4_set_avail_buff(bufidx,size)
int bufidx;
int size;
```

This procedure is used to set the size of buffers in p4's buffer pools. The parameter `bufidx` specifies a particular buffer list, and should be a number from 0 to 7. The `size` parameter specifies that buffers up to that size will be managed by p4 in a particular list. It is important to maintain the buffer sizes in increasing order. The default list of buffer sizes is 64, 256, 1024, 4096, 16384, 65536, 262144, 1048576. This causes wasted space if you send only one large message, causing the allocation of a large buffer which is not reused. Savings in space can be achieved by adjusting these numbers to correspond with the message sizes of your application. If no large messages are sent at all, however, no space is wasted since the large buffers will never be allocated. If you send a message larger that the largest size in this array, p4 will allocate the buffer, and then free it back to the system as soon as it can.

15 Fortran Interface

The Fortran calls to p4 procedures are analogous to their C counterparts, but have Fortran-like names.

```
#include "p4f.h"

p4init()
p4crpg()
p4cleanup()
p4sendr(type,dest,msg,len,rc)
p4sendrx(type,dest,msg,len,data_type,rc)
p4send(type,dest,msg,len,rc)
p4sendx(type,dest,msg,len,data_type,rc)
p4recv(type,from,buf,buflen,msglen,rc)
p4brdcst(type,data,len,rc)
p4brdcstx(type,data,len,data_type,rc)
p4probe(type,from,rc)
p4myclid()
p4nclids()
p4getclids(start,end)
p4myid()
p4clock()
p4ustimer()
p4ntotids()
p4error(str,val)
p4softerrs(new,old)
p4version()
p4flush()
```

```
p4globop(type,x,nelem,size,op,data_type,rc)
   p4intsumop()
   p4intabsmaxop()
   p4intabsminop()
   p4intmaxop()
   p4intminop()
   p4intmultop()
   p4dblsumop()
   p4dblabsmaxop()
   p4dblabsminop()
   p4dblmaxop()
   p4dblminop()
   p4dblmultop()
   p4fltsumop()
   p4fltabsmaxop()
   p4fltabsminop()
   p4fltmaxop()
   p4fltminop()
   p4fltmultop()
```

The `data_type` parameter in the above operations shoudl be one of `P4INT`, `P4LNG`, `P4FLT`, or `P4DBL`. These symbolic constants are defined in the include file '`p4f.h`'.

16 Faster Startup with the Secure Server

P4 processes on remote machines are ordinarily created by `rsh`. For this to work, the user must have permission to create processes on that machine. This permission is normally granted either globally by the system administrator, or locally by the use of '`.rhosts`' files. (See the normal unix man pages under `rhosts`).

Since `rsh` is relatively slow, p4 provides a way to get things started faster. This is accomplished by running the program `serv_p4` in the background on the remote machine. When p4 is creating processes, it will automatically check for the existence of this server and use it if it is running. Remote processes typically start much faster when the server is running. A disadvantage of using the server is that output from `p4_dprintf` is lost unless directed by a command-line argument to a file. (See 7 [Command-Line Arguments], page 12.) When p4 uses `rsh`, the remote process's `stdout` is sent back to the `stdout` of the parent (the p4 master process). We have not yet tested this server on all of the machines that we support. Thus far, we have tested it somewhat on the SYMMETRY, SUN, DEC500, and SGI. We believe that it will work on many other machines, but have not yet verified it on all machines.

An invocation of a set of servers is (currently) associated with a specific port number. This way multiple users can each be running multiple server networks without mutual interference, provided each network of servers is started with a different port number.

To start the secrue server on a machine one can do

```
serv_p4 -d -p <num>
```

where `<num>` is a port number to be associated with a network of servers. If the `-p` option is omitted, the server will pick an unused port number and report

```
Listening on <num>.
```

Then p4 programs to use this network should be started with

```
-ssport <num>
```

The p4 application must also be listed in the user's '`.p4apps`' file in his home directory. This file should be readable only by the user, and should contain the full path names of programs that the user wishes to be startable by the p4 server.

When a p4 master process tries to start a slave process on a remote machine, it will first attempt to do it via the server. If it cannot do so for any reason (no server running, port number mismatch, or program not found in '`.p4apps`' file), then it tries to do so with the remote shell command.

Note that the server is used only to start processes; it plays no role in a p4 computation once the slave processes have been initiated. Rather, a temporary process, called the *listener*, is spawned to manage connection requests that occur during the execution of a p4 program. Neither the server nor the listener consumes any significant amount of CPU time.

There is further discussion of installation options for the servers in the '`README`' file in the '`p4/servers`' subdirectory.

17 Utilities for Managing a p4 Session

A number of useful utilities can be found in the '`bin`' subdirectory. These can be used to start and stop server processes based on the contents of a file of machines one regularly uses, to kill runaway p4 processes in the unlikely case that they cannot or do not terminate automatically when one processes ends abnormally or interrupted from the keyboard, and to merge logfiles created for the use of `upshot` See 18 [Creating Logfiles for Upshot], page 28.

18 Creating Logfiles for Upshot

P4 is distributed with a set of routines for creating logfiles (see '`README`' in the '`p4/alog`' directory. The resulting logfiles can be examined by `upshot`, distributed separately. For details about `upshot`, see [3].

The '`p4/alog`' directory contains a package (ALOG) for creating logs of time-stamped events, that is of general utility, outside of p4. The timestamps are obtained from various microsecond-level resolution timers on various machines. The portable microsecond timing package is contained in the '`usc`' subdirectory. It is used by the ALOG package as well as by the `p4_ustimer` function in p4. Similarly, the ALOG package can be used independently of p4 and `upshot`. Its logfiles were designed to be read and displayed by upshot, but other display packages can be used as well.

18.1 User-Specified Events

The ALOG package consists of a set of macros that can be used to instrument a C program and a set of functions that can be used to instrument a Fortran program. We will focus here primarily on the use of the C interface, which contains more functionality.

The macros that can be used to instrument a program are as follows (from the file `README_ALOG` in the `alog` directory):

```
ALOG_SETUP(pid,flag):
  pid  -  (integer) process id of callee
  flag -  (integer) either ALOG_WRAP or ALOG_TRUNCATE
```

This macro initializes the tracing area for a slave process and must be called once before any event is logged. If the value of `flag` is set to `ALOG_WRAP`, then in the event of no more space for logging events the system will only report the latest n events. If `flag` is set to `ALOG_TRUNCATE` the system will stop logging events as soon as there is no more memory for the events to be logged.

```
ALOG_MASTER(pid,flag):
  pid  -  (integer) process id of the callee
  flag -  (integer) either 0 or 1  (see above)
```

This macro has the same effect over its parameters as `ALOG_SETUP` with the difference that this macro should be referenced by the master process only.

```
ALOG_DEFINE(event,strdef,format):
  event  - (integer) id of event being defined
  strdef - (string) description of 'event'
  format - (string) control string in "printf" format to
```

This macro puts an event definition code into the logfile.

```
ALOG_LOG(pid,event,intdata,strdata):
  pid     - (integer) process id of callee
  event   - (integer) event id to be logged
  intdata - (integer) any integer data for this event
  strdata - (string) any string data (can be the null string)
```

This macro provides the event logging service.

```
ALOG_OUTPUT
  no parameters
```

This macro dumps the events logged into a log file with the name `alogfile.pxx` where `xx` is the logical PID of the callee process. The log file is created in the current directory unless specified otherwise through the macro ALOG_SETDIR.

```
ALOG_SETDIR(dir)
  dir - (string) directory where log file is created
```

This macro sets the output directory for the log file. The default directory for the creation of the log file is the current directory of the process. If used, then this macro MUST be invoked before `ALOG_MASTER/ALOG_SETUP`.

```
ALOG_STATUS(status):
  status - (integer) either ALOG_ON or ALOG_OFF
```

This macro controls the logging status of `ALOG` as follows. Setting `status` to `ALOG_ON` enables logging until it is turned off. Setting `status` to `ALOG_OFF` disables logging until it is turned on again. Logging is enabled at the outset by default.

```
ALOG_ENABLE
  no parameters
```

This macro enables event logging; same as calling `ALOG_STATUS(ALOG_ON)`.

```
ALOG_DISABLE
  no parameters
```

This macro disables event logging; same as calling `ALOG_STATUS(ALOG_OFF)`.

The sample program '`gridlog.shmem.c`' in the '`monitors`' subdirectory contains an example of a program instrumented with ALOG statements. The macro definitions for ALOG are included when you include `#include "p4"` in your program. If the line `#define ALOG_TRACE` is not included before the `#include "p4"`, these macros will generate no code. Thus it is easy to effectively de-instrument the code by recompiling, and there is no need to protect each ALOG statement with an `#ifdef`.

When an ALOG-instrumented program is run, it will produce one logfile for each process. The files will be named '`alogfile.p0`', '`alogfile.p1`', ... These files need to be merged into a single file with the events sored by timestamp. This is accomplished with the program '`mergelogs`', found in the '`bin`' subdirectory. To merge the logfiles, do

```
mergelogs alogfile.p* > myprog.log
rm alogfile.p*
```

The resulting logfile can be examined by upshot or some other logfile examination facility. See [3] for details of the logfile format.

On networks of workstations and some distributed memory machisnes, the microsecond timers on the various processors are synchronized. To produce a usable merged logfile, the '`adjlogs`' program, also found in the '`bin`' directory, can be used to adjust the timestamps for offset and drift before they are merged. For this to work, synchronization events must be placed in the logfiles by an `ALOG_LOG` statement. The event type is then passed to `adjlogs`, which aligns the timestamps, based on the timestamps of the synchonization events. The call to `adjlogs` looks like this, where `<n>` is the type of the synchronization event.

```
adjlogs -e <n>
```

Both `mergelogs` and `adjlogs` are less portable than the other p4 code; you might want to run them on a workstation such as a Sun.

18.2 Examining Log Files with Upshot

`Upshot` is not part of the p4 distribution, but can be obtained from the same anonymous `ftp` location as p4. Take the file '`upshot.tar.Z`' from the directory '`pub/p4`' on `info.mcs.anl.gov`. The distribution contains all necessary documentation on how to install and run `upshot`. It is an X-window program that runs on most workstations. There is no need for a parallel macchine to be involved, once the log files have been obtained.

`Upshot` produces the most interesting displays when certain events (not necessarily all) are defined to be the entry and exit events for certain *states* and then colors are associated with the states. This association is reflected in a *statefile* with a format like the following:

```
1 1 2 red   asking
2 3 4 blue  working
3 5 6 green updating
```

This statefile describes three states. State 1 is defined to be between events 1 and 2. `Upshot` will color it red and label it "asking".

18.3 Automatic Logging of p4 Events

We have found that the most useful events to log and study are those identified by the user and specified in his program. That way he can control the number of events to be logged and the grain size of the states that are represented.

In some cases, however, one wants to study the details of the internal operation of a p4 application, or get some idea of the behavior on one's program without going to the trouble of instrumenting it himself. To this end, p4 itself is instrumented with ALOG statements, although by default they are inactive. To get automatic logging of p4 events (including sending and receiving of each message) one needs first to link to a version of the p4 library that has been compiled with the line `#define ALOG_TRACE` uncommented out in the '`OPTIONS`' file, and secondly, to run with `-p4log` on the command line.

19 Machine-Specific Notes

```
SUN
    (1)  P4 can be installed on this machine with or without SYSV IPC.

HP
    (1)  P4 can be installed on this machine with or without SYSV IPC.
    (2)  Fortran not tested (not avail on our test machine).

DEC5000
    (1)  P4 can be installed on this machine with or without SYSV IPC.

RS6000
    (1)  P4 can be installed on this machine with or without SYSV IPC.

IBM3090
    (1)  P4 can be installed on this machine with or without SYSV IPC.
    (2)  Fortran not supported due to absence of iargc/getarg.
    (3)  There are multiply defined macros in include/rpc/rpc.h.  IBM
         is fixing this in a later OS release.  Meanwhile make your own
         copy and the fix the problem yourself.

TITAN
    (1)  P4 can be installed on this machine with or without SYSV IPC.
    (2)  Fortran not supported due to problems with getting args.

SGI
    (1)  P4 can be installed on this machine with or without SYSV IPC.

NEXT
    (1)  Fortran not supported due to absence of iargc/getarg.

FX2800/FX2800_SWITCH
    (1)  Alliant's switch code not yet ensuring messages
         remain ordered.  p4 currently discovers the switch port for
         the machine it is running on by invoking the internal
         procedure getswport.  This procedure must be customized to
         the installation.

FX8
    (1)  You might need to add MFLAGS = -i to the Makefile

KSR
    (1)  The latest version of the OS produces a link-time error for
         Fortran programs.

IPSC860
    (1)  the script ``runcube'' (in the messages directory) may
         be useful

DELTA
    (1)  the script ``rundelta'' (in the messages directory) may
         be useful

BALANCE
    (1)  Fortran not supported.

SYMMETRY/SYMMETRY_PTX
    (1)  -Z compiler option may be used to control shmalloc/malloc split
    (2)  shared memory message passing not supported in Fortran

TC_2000/TC_2000_TCMP
    (1)  TCMP port not yet complete.
    (2)  For shared-memory execution, one must use cluster ... to
         obtain a private cluster for execution
```

20 Some Common Problems and their Solutions

Our attempt with this manual has been to prevent you from having difficulties. Experience shows that certain common progblems recur, however. In this section we hope to address some of these problems.

"Permission Denied." p4 slave processes are started by forks (for slaves in the same shared-memory cluster), by the server, or by the remote shell command. If the server is running on the target machine then that must be configured to allow remote processes to be started. To test whether this is your problem, try

```
rsh target.machine date
```

If you still get the "Permission denied." message, then the problem has nothing to do with p4. See `hosts.equiv` or `.rhosts` in the system man pages.

"More processes than message queues" Under the default configuration of p4, uniprocessors, such as most workstations, cannot have multiple process sharing memory. Thus your procgroup file for a workstation network should always look like

```
      local 0
machine1 1 pathname
machine2 1 pathname
machine3 1 pathname
    .
    .
```

The "local" means "only the master on the startup machine; no local slaves sharing memory".

It is possible, at some cost in message-passing efficiency, to have a cluster of processes sharing memory on a workstation, but in this case p4 must have been installed with the `SYSV_IPC` option set in the '`OPTIONS`' file. The cost is that a process waiting for a message must spin between checking for a message arriving on a socket and a message arriving through shared memory.

"p4_error: local is not first entry in procgroup" the first line of the procgroup file must be the "local" entry, specifying the number of slaves that will be run on the master machine in addition to the master process.

"gethostbyname failed 100 times" Check for an invalid machine name in the procgroup file.

"pgm_path_name: Command not found" P4 tried to start the program with the given name on a remote machine and the program did not exist. Verify the full path name of the program.

program hangs You may have failed to initialize the `type` and `from` fields before a `p4_recv`. You might have used `p4_sendr` between two processes at the same time, which will deadlock if you think about it, or even if you don't. Use `p4_send` instead.

program hangs or has bad data in received message You might have failed to set the pointer to the incoming buffer to NULL, or to have specifically allocated a buffer with `p4_msg_alloc`, before a `p4_recv`.

program ignores command-line arguments You might have passed `argc` instead of `&argc` to `p4_initenv`.

program runs out of memory You may need to call `p4_msg_free` after each `p4_recv`, or reuse buffers by pre-allocating them.

21 Concept Index

A

allocating buffers . . . 17
askfor monitor . . . 21
automatic logging . . . 31

B

barrier . . . 18, 21
buffer pool management . . . 26

C

clocks . . . 22
cluster management . . . 15
command-line arguments . . . 12

D

data types for monitors . . . 19
deallocating buffers . . . 10, 17
debugging . . . 23
directory structure . . . 2
documentation . . . 4

E

emacs Info . . . 4
examples . . . 5

F

faster startup . . . 27
Fortran interface . . . 26

G

getting started . . . 5
global operations . . . 17

I

installation . . . 3
interrupts . . . 25
IPC . . . 33

L

lock . . . 20
logging . . . 28

M

machines . . . 3
man pages . . . 4
message-passing functions . . . 15
monitor data types . . . 19
monitor primitives . . . 19
monitors . . . 20

O

online help . . . 4
OPTIONS . . . 2

P

permission denied . . . 33
permissions . . . 27
problems . . . 33
process management . . . 13
procgroup files . . . 6

R

receiving messages . . . 16
return codes . . . 13
rhosts . . . 27
runaway processes . . . 25

S

sending messages . . . 16
servers . . . 27, 28
shared memory data types . . . 19
shared memory functions . . . 19
slave subroutine . . . 9
statefile . . . 30

T

testing . . . 4
timestamps . . . 28
timing . . . 22
tracefiles . . . 28

U

unlock . . . 20
upshot . . . 28, 30
user events . . . 29
utilities . . . 28

22 Function Index

A

ALOG_DEFINE . . . 29
ALOG_DISABLE . . . 30
ALOG_ENABLE . . . 30
ALOG_LOG . . . 29
ALOG_MASTER . . . 29
ALOG_OUTPUT . . . 29
ALOG_SETDIR . . . 29
ALOG_SETUP . . . 29
ALOG_STATUS . . . 29

P
p4_alloc_procgroup 14
p4_am_i_cluster_master 15
p4_askfor 21
p4_askfor_init 22
p4_barrier 21
p4_barrier_init 21
p4_broadcast 17
p4_broadcastx 17
p4_clock 22
p4_create 13
p4_create_procgroup 13
p4_dprintf 23
p4_dprintfl 23
p4_error 24
p4_free 19
p4_get_cluster_ids 15
p4_get_cluster_masters 25
p4_get_dbg_level 23
p4_get_my_cluster_id 15
p4_get_my_id 14
p4_getsub 20
p4_getsub_init 21
p4_getsubs 21
p4_global_barrier 18
p4_global_op 17
p4_initenv 13
p4_lock 20
p4_lock_init 20
p4_malloc 19
p4_mcontinue 20
p4_mdelay 20
p4_menter 19
p4_messages_available 16
p4_mexit 20
p4_moninit 19
p4_msg_alloc 17
p4_msg_free 17
p4_num_cluster_ids 15
p4_num_total_ids 14
p4_num_total_slaves 15
p4_print_avail_buffs 25
p4_probend 22
p4_progend 22
p4_recv 16
p4_send 16
p4_sendb 16
p4_sendbr 16
p4_sendbrx 16
p4_sendbx 16
p4_sendr 16
p4_sendrx 16
p4_sendx 16
p4_set_avail_buff 26
p4_set_dbg_level 23
p4_shfree 19
p4_shmalloc 19
p4_soft_errors 25
p4_startup 14
p4_unlock 20
p4_update 22
p4_usrollover 23
p4_ustimer 23
p4_version 25
p4_wait_for_end 14

References

[1] J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens. *Portable Programs for Parallel Processors.* Holt, Rinehart, and Winston, 1987.

[2] Per Brinch Hansen. *The Architecture of Concurrent Programs.* Prentice-Hall, Inc., 1977.

[3] Virginia Herrarte and Ewing Lusk. Studying parallel program behavior with `upshot`. Technical Report ANL–91/15, Argonne National Laboratory, Argonne, IL 60439, 1991.

[4] C. A. R. Hoare. Monitors: an operating system structuring concept. *Comunications of the ACM*, pages 549–557, October 1974.