Paragon ™
System

User's Guide

April 1996

Order Number: 312489-005

Paragon™ System User's Guide

Intel® Corporation

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update or to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

WARNING

Some of the circuitry inside this system operates at hazardous energy and electric shock voltage levels. To avoid the risk of personal injury due to contact with an energy hazard, or risk of electric shock, do not enter any portion of this system unless it is intended to be accessible without the use of a tool. The areas that are considered accessible are the outer enclosure and the area just inside the front door when all of the front panels are installed, and the front of the diagnostic station. There are no user serviceable areas inside the system. Refer any need for such access only to technical personnel that have been qualified by Intel Corporation.

CAUTION

This equipment has been tested and found to comply with the limits for a Class A digital device, pursuant to Part 15 of the FCC Rules. These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment. This equipment generates, uses, and can radiate radio frequency energy and, if not installed and used in accordance with the instruction manual, may cause harmful interference to radio communications. Operation of this equipment in a residential area is likely to cause harmful interference in which case the user will be required to correct the interference at his own expense.

LIMITED RIGHTS

Preface

This manual tells how to use the operating system on a Paragon supercomputer.

This manual assumes that you are an application programmer proficient in the C or Fortran language and the UNIX operating system. The manual provides you with enough detail to begin using your system.

NOTE

In this manual, "operating system" refers to the operating system that runs on the nodes of the Paragon(TM) supercomputer.

NOTE

Programming examples in this manual are intended to demonstrate how to use the system calls provided by the operating system, but are not necessarily examples of good programming practice.

For example, in some cases, the return values of functions are not checked for error conditions. This is not recommended, but the error checks have been omitted in order to make the example shorter and easier to read.

Organization

Chapter 1 Provides an overview of the operating system software and Paragon supercomputer hardware.

Chapter 2 Describes the operating system commands that you can enter at the shell prompt and the operating system cross-development commands that run on supported workstations.

Chapter 3 Describes the message-passing system calls available to programs in operating system.

Chapter 4 Describes the other general-purpose system calls available in operating system.

Chapter 5 Describes the parallel I/O calls you can use for parallel access to the Paragon supercomputer's file systems.

Chapter 6 Describes SMP programming model and the *pthreads* package. The pthreads package lets you create and control multiple threads (also called "lightweight processes") within your programs.

Chapter 7 Tells how to prepare an application for the operating system operating system. The steps described are applicable to applications that are written for a parallel computer and applications that are ported from a sequential computer. This chapter discusses three examples: an integration, a matrix*vector multiplication, and the N-Queens problem.

Chapter 8 Presents some techniques you can use to improve the performance of your parallel applications.

Appendix A Summarizes the commands and system calls of operating system. The complete syntax of each command and call is provided, along with a brief description of each.

Appendix B Describes the level of support offered by operating system for the commands and system calls of the iPSC® system.

Notational Conventions

This manual uses the following notational conventions:

Bold Identifies command names and switches, system call names, reserved words, and other items that must be used exactly as shown.

Italic Identifies variables, filenames, directories, partitions, user names, and writer annotations in examples. Italic type style is also occasionally used to emphasize a word or phrase.

`Plain-Monospace`
Identifies computer output (prompts and messages), examples, and values of variables.

Bold-Italic-Monospace
Identifies user input (what you enter in response to some prompt).

Bold-Monospace
Identifies the names of keyboard keys (which are also enclosed in angle brackets). A dash indicates that the key preceding the dash is to be held down *while* the key following the dash is pressed. For example:

<Break> **<s>** **<Ctrl-Alt-Del>**

[ ] (Brackets) Surround optional items.

... (Ellipsis dots) Indicate that the preceding item may be repeated.

| (Bar) Separates two or more items of which you may select only one.

{ } (Braces) Surround two or more items of which you must select one.

Applicable Documents

For more information, refer to the following manuals. See the *Paragon™ System Technical Documentation Guide* for information on the complete Paragon document set and ordering information.

Paragon™ Manuals

- *Paragon™ System Commands Reference Manual*
- *Paragon™ System Network Queueing System Manual*
- *Paragon™ System C Compiler User's Guide*
- *Paragon™ System Fortran Compiler User's Guide*
- *Paragon™ System C Calls Reference Manual*
- *Paragon™ System Fortran Calls Reference Manual*
- *Paragon™ System Application Tools User's Guide*
- *Paragon™ System Interactive Parallel Debugger Reference Manual*
- *Paragon™ System Administrator's Guide*

Other Manuals

- *OSF/1 User's Guide*
- *OSF/1 Programmer's Reference*
- *OSF/1 Command Reference*
- *Effective Fortran 77* - Michael Metcalf
- *C: A Reference Manual* - Harbison and Steele
- *The C Programming Language* - Kernighan and Ritchie
- *CLASSPACK Basic Math Library User's Guide* - Kuck & Associates
- *CLASSPACK Basic Math Library/C User's Guide* - Kuck & Associates

Comments and Assistance

Intel Scalable Systems Division is eager to hear of your experiences with our products. Please call us if you need assistance, have questions, or otherwise want to comment on your Paragon system.

U.S.A./Canada Intel Corporation
Phone: 800-421-2823
Internet: support@ssd.intel.com

France Intel Corporation
1 Rue Edison-BP303
78054 St. Quentin-en-Yvelines Cedex
France
0590 8602 (toll free)

Intel Japan K.K.
Scalable Systems Division
5-6 Tokodai, Tsukuba City
Ibaraki-Ken 300-26
Japan
0298-47-8904

United Kingdom Intel Corporation (UK) Ltd.
Scalable Systems Division
Pipers Way
Swindon SN3 IRJ
England
0800 212665 (toll free)
(44) 793 491056
(44) 793 431062
(44) 793 480874
(44) 793 495108

Germany Intel Semiconductor GmbH
Dornacher Strasse 1
85622 Feldkirchen bei Muenchen
Germany
0130 813741 (toll free)

World Headquarters
Intel Corporation
Scalable Systems Division
15201 N.W. Greenbrier Parkway
Beaverton, Oregon 97006
U.S.A.
(503) 677-7600 (Monday through Friday, 8 AM to 5 PM Pacific Time)
Fax: (503) 677-9147

If you have comments about our manuals, please fill out and mail the enclosed Comment Card. You can also send your comments electronically to the following address:

techpubs@ssd.intel.com
(Internet)

Table of Contents

Chapter 1 Introduction

Chapter 2
Using System Commands

Chapter 3
Using Message-Passing System Calls

Chapter 4
Managing Applications and Partitions with System Calls

Chapter 5
Using Parallel File I/O

Chapter 6
SMP Programming

Chapter 7 Designing a Parallel Application

Chapter 8 Improving Performance

Appendix A Summary of Commands and System Calls

Appendix B
iPSC® System Compatibility

List of Illustrations

List of Tables

Introduction 1

Introduction

This chapter introduces the operating system and the hardware it runs on.

In a Paragon supercomputer, a large number of *nodes* work concurrently on the parts of a problem. Each node has multiple processors and can run multiple processes; each process can have multiple *threads*. The processes and threads on each node time-share the node's processors, using the standard OSF/1 scheduling mechanisms. Each process can be a stand-alone program (such as a shell, compiler, or editor), or can be part of a *parallel application.*

A parallel application consists of a group of closely related processes that work together on a single problem. They synchronize their actions and share information by passing *messages*, which are created and controlled by special operating system system calls.

The processes in an application can also share disk files; operating system *parallel I/O calls* insure that access to these files is efficient and properly synchronized.

System Hardware

The operating system runs on several models of Paragon supercomputers. These systems all have a large number of *nodes* connected by a high-speed *node interconnect network*, and a number of *I/O interfaces* to communicate with the outside world.

Nodes

Each node is essentially a separate computer, with multiple i860® processors and 16M bytes or more of memory. Nodes can run distinct programs and have distinct memory spaces. They can team up to work on the same problem and exchange data by passing messages. A Paragon supercomputer can have up to more than 2000 nodes. Each node can run more than one process at the same time; these processes can belong to the same or different applications.

There are three kinds physical nodes:

MP node A three-processor node that runs with two processors as general processors and one processor as message coprocessor.

GP node A two-processor node that runs with one processor as a general processor and one processor as a message coprocessor.

MP-as-GP node A three-processor node that runs with one processor as a general processor, one processor as a message coprocessor, and one processor turned off.

An MP system is a Paragon supercomputer that is configured with MP nodes only. A GP system is a Paragon supercomputer that is configured with either GP nodes or MP-as-GP nodes only. In a GP system, MP nodes are automatically configured as MP-as-GP nodes.

The system administrator can choose to dedicate some nodes to interactive processes, such as shells and editors, and other nodes to compute-intensive applications. The nodes used for interactive processes are called *service nodes*, and the nodes used for compute-intensive applications are called *compute nodes*.

Node Interconnect Network

The nodes are connected by a high-speed *node interconnect network*. Each node interfaces to this network through special hardware that monitors the network and extracts only those messages addressed to its attached node. Messages addressed to other nodes are passed on without interrupting the node processor. For most applications, you can think of each node as being fully connected to all the other nodes.

I/O Interfaces

Some nodes are equipped with a SCSI interface, Ethernet interface, or other I/O connection. These nodes manage the system's disk and tape drives, network connections, and other I/O facilities. Nodes with I/O interfaces communicate with the other nodes over the node interconnect network. However, this access is transparent: processes on nodes without I/O hardware access the I/O facilities using standard OSF/1 system calls, just as though they were directly connected. Nodes with I/O interfaces are otherwise identical to nodes without I/O interfaces, and can run user processes.

Front Panel LEDs (Paragon™ XP/S System Only)

On the Paragon XP/S System system, each cabinet has a number of Light-Emitting Diodes (LEDs) on its front panel that inform you of the status of the system, the nodes, and messages between nodes. The front panel LEDs are shown in Figure 1-1.

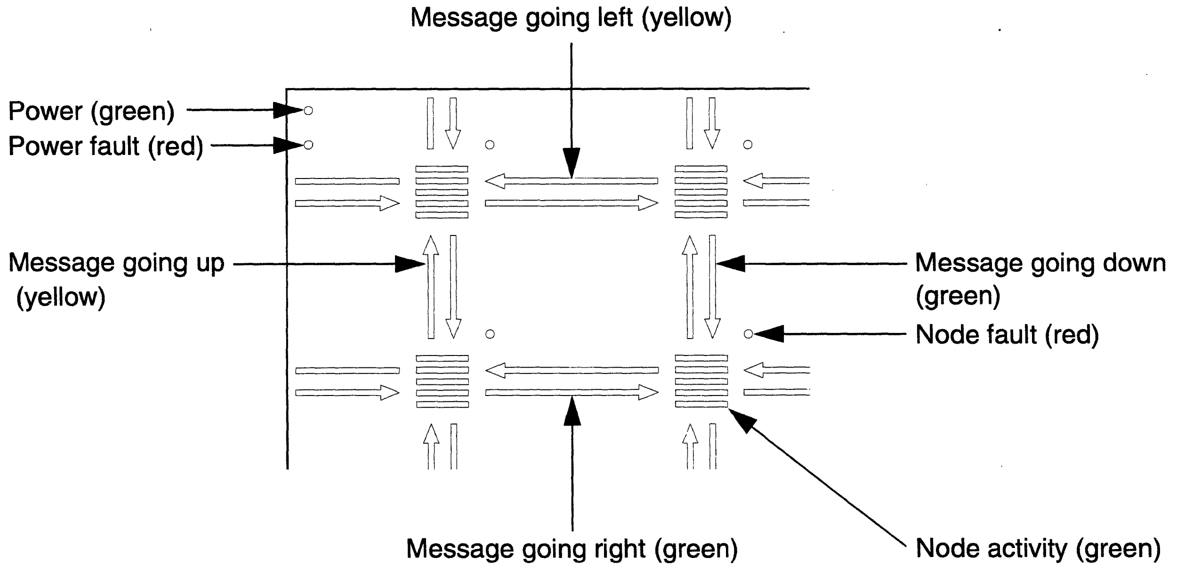

Message going left (yellow)
Power (green)
Power fault (red)
Message going up (yellow)
Message going down (green)
Node fault (red)
Message going right (green)
Node activity (green)

Figure 1-1. Front Panel LEDs (Paragon™ XP/S System Only)

Each cabinet has four LED panels, each of which shows the status of 16 nodes in a 4 by 4 grid. Figure 1-1 shows the upper left corner of one LED panel. The meanings of the LEDs are as follows:

- The round green LED in the upper left corner of the top LED panel in each cabinet indicates that power has been supplied to the cabinet. (The corresponding LEDs in the other three panels never illuminate.)

- The round red LED just below the green power LED indicates a fault in the cabinet's power subsystem. If a fault is detected by the cabinet's self-tests, this LED illuminates. (The corresponding LEDs in the other three panels never illuminate.)

- The square groups of horizontal green LED bars show the amount of computational activity on the nodes. Each group represents one node. The more active a node is, the more green LEDs are illuminated, in a bar graph moving out from the center. Figure 1-2 shows the six possible ways these LEDs can be illuminated, showing activity levels from 0% to 100%.

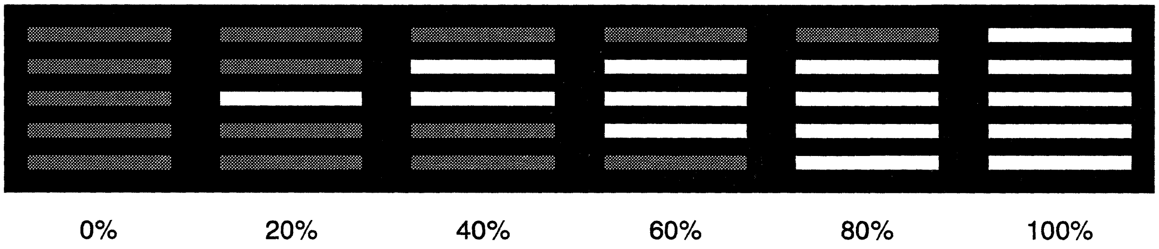


Figure 1-2. Node Activity LEDs

- The arrow-shaped yellow and green LED bars indicate messages. When a message is passed from one node to another, all the arrow LEDs along its path illuminate. (Messages always travel first in the X direction (horizontally), then in the Y direction (vertically). Messages never change direction more than once.) Yellow arrows show messages going up or to the left; green arrows show messages going down or to the right. When the arrows are illuminated, a light pattern moves along the arrow to show the direction of motion.

- The round red LED associated with each node indicates a hardware fault on the node. If a fault is detected by the node's self-tests, the red LED illuminates.

System Software

The nodes run the operating system, which is based on the OSF/1 operating system from the Open Software Foundation. The same operating system runs on every node. OSF/1 is a version of the UNIX operating system that supports most industry standards; operating system is an extended version of OSF/1 with enhancements to support parallel processing.

The Paragon supercomputer also comes with a *cross-development facility*, which you can use to compile and link operating system programs on supported workstations.

Operating System

The operating system provides all the standard features of the OSF/1 operating system, with extensions to provide a *single system image* across multiple nodes. This single system image makes all the nodes appear to be one large system. For example, all the nodes share a single file system, all the nodes have equal access to the system's I/O devices, and process identifiers (PIDs) are unique throughout the system. A process on one node can pipe its output to a process on another node, and the command **kill** *pid* on any node kills the specified process, no matter which node the process is running on.

The single system image does *not* combine all the nodes' memory into a single address space. Rather, each process has its own address space. The physical memory available to each process is limited to the memory of the node on which it is running. However, because OSF/1 provides *virtual memory*, a process's address space can be up to 2G bytes in size; memory pages that do not fit in physical memory are paged to disk. As in most multi-user systems, the address spaces of the different processes on the system are completely independent, unless two or more processes make special shared virtual memory calls to explicitly share part of their memory.

In addition to the standard facilities of OSF/1, the operating system provides message passing capability, Parallel File System access, and various other utilities to programs running on the Paragon supercomputer. With operating system calls, your programs can perform the following functions:

- Exchange messages with processes running on other nodes (or the same node).
- Read and write files on the Parallel File System.
- Perform 64-bit integer arithmetic.
- Find out information about the computing environment.
- Perform global operations.
- Create and control parallel applications and partitions.

User Model

The operating system is a complete implementation of OSF/1, and provides a full range of services, commands, and system calls. It has its own file system, shells, compilers, editors, network connections, and all the other features needed in a stand-alone computer system. It also supports NFS, the Network File System, so it can share data with other systems on your network. You can edit and compile programs, send and receive mail, read online manual pages, and do all your other daily work on the Paragon supercomputer.

You access the Paragon supercomputer by logging into a separate computer (typically your UNIX workstation) and then connecting to the Paragon supercomputer over a local-area network, using a command such as **rlogin** or **telnet**. The Paragon supercomputer does not have any dedicated hardware terminals.

You compile and link your application with the self-hosted operating system compilers and linker. You then execute your application on the nodes of the Paragon supercomputer simply by typing the application's name on the shell command line. Command-line switches, or arguments to system calls in the program, determine the number of nodes on which the application executes.

When you run an application, it runs in a *partition.* A partition is a group of nodes with an associated set of parameters that controls some of the run-time characteristics of the applications within it. You can use commands or system calls to create, modify, and remove partitions. However, the operations you are allowed to perform on your system's partitions may be restricted by the policies of your site.

The operating system operating system also provides a suite of program development tools, such as a debugger, profiler, and parallel performance analysis tools. These tools are described in the *Paragon™ System Application Tools User's Guide.*

Programming Model

The most common programming model used with operating system is the "single program, multiple data" (SPMD) model. In this model, the same program runs on each node in the application, but each node works on only part of the data.

- For some problems, called "perfectly parallel" problems, each node can do its work without access to data held by other nodes. In this case, each node operates completely independently.

- For other types of problems, each node needs data from other nodes to do its work. In this case, the nodes can share data by passing messages. Messages can also be used to synchronize node operations.

Because each node is an independent computer, you can also use other programming models. One example is the "manager-worker" model, in which one "manager" program starts up several "worker" programs on other nodes, then gathers and interprets their results.

Cross-Development Facility

The operating system comes with a complete program development environment, including compilers, linker, libraries, and related tools. You can perform all phases of program development on the Paragon supercomputer. In addition, the compilers, linker, and libraries are available on selected UNIX workstations for *cross-development.* This lets you edit, compile, and link operating system programs on your own workstation, then download your application to your Paragon supercomputer where you run the application.

The cross-development facility does not include a way to run a operating system executable that resides on your workstation's disk. You must transfer your executable files to the Paragon supercomputer for execution and debugging. You can do this by mounting your workstation's file system onto the Paragon supercomputer, or the Paragon supercomputer's file system onto your workstation, using the Network File System (NFS). You can also use commands such as **rcp** or **ftp** to copy the executable files to the Paragon supercomputer. To execute files on the Paragon supercomputer once they are transferred, you can use the standard **rsh** or **rcmd** command from your workstation.

Using System Commands 2

Introduction

This chapter tells you how to use operating system commands to perform the following tasks:

- Compiling and linking applications.

- Running applications.

- Managing running applications.

- Managing partitions.

The commands discussed in this chapter are available to all users. See the *Paragon*™ *System Administrator's Guide* for information on commands that require root privilege.

This chapter does *not* discuss NQS, the Network Queueing System, which is used at some sites to schedule application execution. See the *Paragon*™ *System Network Queueing System Manual* for information on NQS.

Terminology

This chapter uses the following terms:

- A *parallel application*, usually just called an *application* in this manual, is a group of cooperating processes that runs on the nodes of the Paragon supercomputer.

- A *program* is a file (source or executable). An application consists of one or more programs running on one or more nodes. The term *program* is also used to refer to a non-parallel program (an ordinary program that runs on one node).

- A *partition* is a named group of nodes. When you run a parallel application, you must select a partition to run it in (if you don't, it runs in your *default partition*). The partition places limits on some of the execution characteristics of the application, such as how many nodes it can use and how long it can use them before it is "rolled out" and another application is "rolled in." You can allocate all of the nodes of the partition to the application, or just some of them. This allocation may or may not be exclusive, depending on the characteristics of the partition.

  All Paragon supercomputers have two special partitions called the *service partition* and the *compute partition*. The service partition is used to run non-parallel programs such as shells and editors, and the compute partition is used to run parallel applications. The other partitions on your system, and what you can do with them, are determined by your system administrator.

Using Commands on the Paragon™ Supercomputer

The operating system provides all of the standard commands of OSF/1, such as **cat** and **ls**, which work as specified by the Open Software Foundation. These commands are not described in this chapter; see the *OSF/1 Command Reference* for information on these commands.

The operating system also provides several commands that are not specified by the Open Software Foundation, such as **mkpart** and **rmpart**. These commands are described in this chapter, and manual pages for these commands are provided in the *Paragon™ System Commands Reference Manual.*

To use any of these commands, you must first log into an Paragon supercomputer. Paragon supercomputers have no directly-attached terminals; you must first log into another system (typically a workstation running some variant of the UNIX operating system) and then log into the Paragon supercomputer over the network, using a command such as **rlogin** or **telnet**. Once you have logged in, you use these commands in the same way as commands on any other computer running OSF/1.

Using Commands on Workstations

The operating system also comes with several commands that run on workstations (for example, the **icc** and **if77** cross-compilers). These commands are described briefly in this chapter; complete descriptions and manual pages for these commands are provided in the *Paragon™ System C Compiler User's Guide* and *Paragon™ System Fortran Compiler User's Guide.*

To use these commands, you must first log into a workstation on which these commands are supported, then configure your account as described under "Configuring Your Environment for Cross-Development" on page 2-6. Once you have done this, you can use the operating system cross-development commands in the same way as other commands on the workstation. However, if you compile an application on a workstation you must transfer the executable file to a Paragon supercomputer to execute it. Depending on your local configuration, you may be able to use the

Network File System (NFS), the **rcp** command, the **ftp** command, or some other technique to do this. Ask your system administrator about how files are shared between the Paragon supercomputer and other systems on your network.

A Quick Example

Here is a quick example that shows you how to compile, link, and execute a simple application on a Paragon supercomputer.

Information You Need

Before you begin, you will need the following information:

- The network name of your Paragon supercomputer.
- The command to use to log into the Paragon supercomputer, such as **rlogin** or **telnet**.
- Your user name and password on the Paragon supercomputer (if necessary).
- The name of the *default partition* you should use to run parallel applications.

This information should be available from your system administrator.

Compiling, Linking, and Executing an Application

Once you have the necessary information, the procedure to compile, link, and execute an application is as follows:

1. Log into the Paragon supercomputer, as instructed by your system administrator.
2. Set the environment variable *NX_DFLT_PART* to the name of your default partition:
   - If you use the C shell, use the following command:

     ```
     % setenv NX_DFLT_PART partition_name
     ```

   - If you use the Bourne or Korn shell, use the following commands:

     ```
     $ NX_DFLT_PART=partition_name
     $ export NX_DFLT_PART
     ```

3. Type in a short program:

- If you are a Fortran programmer, type the following program into the file *myapp.f*:

```
      program hello
      include 'fnx.h'

      write(*,100) mynode()
100   format('Hello from node', i4, '!')

      end
```

- If you are a C programmer, type the following program into the file *myapp.c*:

```
#include <nx.h>

main()
{
    printf("Hello from node %d!\n", mynode());
}
```

4. Compile the program into an executable file:

   - If you are a Fortran programmer, use the following command:

```
% f77 -nx -o myapp myapp.f
```

   - If you are a C programmer, use the following command:

```
% cc -nx -o myapp myapp.c
```

5. Execute the resulting file, *myapp*, on four nodes with the following command:

```
% myapp -sz 4
Hello from node 0!
Hello from node 3!
Hello from node 1!
Hello from node 2!
```

   The order in which the output lines appear may vary.

That's all there is to it! Of course, the operating system provides many additional commands and switches you can use to control the behavior of the compiler and the resulting application. These commands and switches are described in the rest of this chapter.

Compiling and Linking Applications

Command Synopsis	Description
cc -nx [*switches*] *sourcefile...*	Compile an application written in C on a Paragon supercomputer.
f77 -nx [*switches*] *sourcefile...*	Compile an application written in Fortran on a Paragon supercomputer.
icc -nx [*switches*] *sourcefile...*	Compile an application written in C on a Paragon supercomputer or cross-development workstation.
if77 -nx [*switches*] *sourcefile...*	Compile an application written in Fortran on a Paragon supercomputer or cross-development workstation.

You can compile and link applications on the Paragon supercomputer itself, or on a workstation that supports the operating system cross-development environment. On the Paragon supercomputer, you can use the "native" commands **cc** and **f77** or the "cross-development" commands **icc** and **if77**. On a workstation, you must use the cross-development commands **icc** and **if77**. The native and cross-development versions of each command take the same switches and work identically.

When compiling and linking an application, you should generally use the switch **-nx** on the command line. The **-nx** switch has three effects:

- If used while linking a C or Fortran program, it links in *libnx.a*, the library that contains all the system calls described in this manual.

- If used while linking a C or Fortran program, it links in a special start-up routine that starts up the program on multiple nodes, as specified by standard command line switches and environment variables.

- If used while compiling a C program, it defines the preprocessor symbol **__NODE**. The program being compiled can use preprocessor statements such as **#ifdef** to control compilation based on whether or not this symbol is defined. (This preprocessor symbol is *not* defined if **-nx** is used while compiling a Fortran program.)

For example, the following command line compiles and links the file *myapp.c* to create an executable file called *myapp* (on the Paragon supercomputer):

```
% cc -nx -o myapp myapp.c
```

The following command line has the same effect (on the Paragon supercomputer or a cross-development workstation):

```
% icc -nx -o myapp myapp.c
```

NOTE

Do not use **-nx** if your application uses any of the **nx_initve...()** calls.

The operating system provides the **nx_initve...()** calls and related functions to give your application more control over the way it starts up. They let the application perform actions for itself that are normally performed for it by **-nx**. If you link your application with **-nx** and it also makes any **nx_initve...()** call itself, the application's call to **nx_initve...()** will fail and return -1. See "Managing Applications" on page 4-2 for more information on **nx_initve...()** and related functions.

To link an application that calls **nx_initve...()**, use the switch **-lnx** instead of **-nx**. The **-lnx** switch links in *libnx.a*, but without the special start-up routine supplied by **-nx**. A program linked with **-lnx** can use all the calls described in this manual, but does not automatically start itself on multiple nodes. (Note that the **-lnx** switch must appear on the compiler command line *after* the filenames of any source or object files that use these calls.) Note that the preprocessor symbol **__NODE** is *not* defined by **-lnx**.

A program that is not linked with **-nx** *and* does not call **nx_initve...()** is not a parallel application. It does not recognize the command-line switches described under "Running Applications" on page 2-11, and it always runs on one node in the service partition. (If it creates additional processes by calling **fork()**, they may run on the same node or a different node, but they will always run in the service partition.)

Configuring Your Environment for Cross-Development

Before you can use the **icc** and **if77** commands on your workstation, you must configure your environment as follows:

- The environment variable *PARAGON_XDEV* must be set to the pathname of the directory that contains the operating system cross-development facility. If you don't know this pathname, ask your system administrator.

- Your execution search path (*PATH* or *path* variable) must include the directory *$PARAGON_XDEV/paragon/bin.arch*, where *arch* identifies the architecture of your workstation (such as **sun4** for a Sun-4 workstation).

- If you want to read online manual pages on your workstation, your online manual page search path (*MANPATH* variable or equivalent facility) must include the directory *$PARAGON_XDEV/paragon/man.*

You should put the definitions of these variables into your *.cshrc* or *.login* file (or the equivalent start-up file for your shell). For example, suppose the operating system cross-development facility is installed in the directory */usr/local/XDEV*. If you use the C shell, you would add these lines to your *.cshrc* file:

```
setenv PARAGON_XDEV /usr/local/XDEV
set path=( $path $PARAGON_XDEV/paragon/bin.'arch' )
setenv MANPATH "${MANPATH}:${PARAGON_XDEV}/paragon/man"
```

(The curly braces in `"${MANPATH}:${PARAGON_XDEV}/paragon/man"` are necessary because a colon after a variable name is special to the C shell.)

Once your environment is properly configured, you can use the **icc** or **if77** command to compile and link applications on your workstation. For example, the following command line compiles and links the file *myapp.f* to create an executable file called *myapp*:

```
% if77 -nx -o myapp myapp.f
```

The executable file, *myapp*, can only be executed on the Paragon supercomputer. You can do this by putting it in a directory that is shared between your workstation and the Paragon supercomputer with the Network File System (NFS), or by copying it to the Paragon supercomputer with the **ftp** or **rcp** command. If you use the **ftp** command, the resulting file may not have execute permission; if this happens, use the **chmod** command on the Paragon supercomputer to give *myapp* execute permission.

NOTE

The Paragon system versions of the compilers are not the same as their iPSC® system equivalents.

If you develop programs for the iPSC series of supercomputers from Intel Corporation as well as for the Paragon system, you must be sure that your execution search path (*PATH* or *path* variable) is set appropriately for your current target system. To compile a program for the Paragon system, the variable *PARAGON_XDEV* must be set appropriately and your execution search path must include *$PARAGON_XDEV/paragon/bin.arch*; to compile a program for the iPSC system, the variable *IPSC_XDEV* must be set appropriately and your execution search path must include *$IPSC_XDEV/i860/bin.arch* instead. Be sure that your execution search path does not include both these directories at the same time.

Tips for Compiling and Linking

The following sections give you some tips for compiling and linking applications (on either the Paragon supercomputer or a cross-development workstation).

Using Other Switches

The **cc**, **f77**, **icc**, and **if77** commands have a variety of switches to control their operation. For a description of these switches and other information on these commands, see the online manual pages for the commands or the following printed manuals:

cc, icc	*Paragon™ System C Compiler User's Guide.*
f77, if77	*Paragon™ System Fortran Compiler User's Guide.*

Including *nx.h* or *fnx.h*

As a general rule, always include the file *nx.h* in all C programs and the file f*nx.h* in all Fortran programs. These files contain definitions and declarations needed by the operating system's system calls. Although a specific application may not need the definitions and declarations in these include files, the overhead involved in including them in all C or Fortran programs is minor. Include *nx.h* in your C programs as follows:

```
#include <nx.h>
```

Include *nx.h* in your Fortran programs as follows

```
include 'fnx.h'
```

Specifying Include File and Library Pathnames

The standard include and library directories depend on whether you are using the native development commands or the cross-development commands:

- The native development commands search for include files in the directory */usr/include*, and they search for libraries in the directories */usr/ccs/lib* (searched first) and */usr/lib* (searched second).

- The cross-development commands search for include files in the directory *$PARAGON_XDEV/paragon/include*, and they search for all libraries in the directory *$PARAGON_XDEV/paragon/lib-coff.*

Note, though, that on the Paragon supercomputer the directories */usr/paragon/XDEV/paragon/lib-coff* and */usr/ccs/lib* are identical, the directories */usr/paragon/XDEV/paragon/include* and */usr/include* are identical, and the default for *$PARAGON_XDEV* is */usr/paragon/XDEV*, so this difference may not be significant.

If you need to include a file that is not in the standard include directory or in the same directory as the source file, you must use the **-I** switch on the compiler command line to identify the nonstandard directory. For example, the following command line compiles and links an application that uses include files in the directory */usr/local/include*:

```
% icc -nx myapp.c -I/usr/local/include
```

If you need to link to a library that is not in one of the standard library directories, then you must modify the command line in one of the following ways:

- Use the **-L** switch to provide the pathname of the directory in which the library is located. For example, the following command line compiles and links an application that depends on the library *libfft.a* located in the directory */usr/local/lib*:

  ```
  % icc -nx -L/usr/local/lib myapp.c -lfft
  ```

- Specify the complete pathname of the appropriate library or libraries on the command line. For example, the following command line also compiles and links an application that depends on the library *libfft.a* located in the directory */usr/local/lib*:

  ```
  % if77 -nx myapp.c /usr/local/lib/libfft.a
  ```

Preprocessing a Fortran Program

If your Fortran program is in a file whose filename ends with an uppercase ".F" (rather than the standard lowercase ".f"), the **if77** command runs a preprocessor (like the standard C preprocessor) on the file. This enables you to use lines like the following in a Fortran program:

```
#include <file.h>

#define MAX 87
```

Order of Switches

Most **cc**, **f77**, **icc**, and **if77** switches are not order-sensitive. However, order is important for the **-I**, **-L**, and **-l** switches and for listing libraries when linking. When constructing command lines, keep the following guidelines in mind:

- List include directories (**-I** switch) in the order in which they should be searched. The list of include directories you specify with **-I** switches is collected together and used for all source files you specify. For example, the following command looks for include files in the directory *myincludes*, then the directory *../includes*, and finally the standard include directory when compiling *a.c*, *b.c*, and *c.c*:

```
% icc a.c -Imyincludes b.c -I../includes c.c
```

- List libraries in the order in which they should be searched. The system's linkers are single-pass linkers; they cannot resolve a backward library reference (i.e., a reference to a library object that was defined in a library that has already been searched). Note that this means that if you use the **-lnx** switch, you should place it *after* any source files that need it, as follows:

```
% if77 -o myapp myapp.f -lnx
```

  Backward references between objects (*.o* files), however, are not a problem, as all listed objects are linked unconditionally.

- The **-L** switch affects only the search path of libraries that are listed *after* the **-L** switch. For example, the following command searches only the standard library directories for the library *libnews.a*, but searches the directory *../mylibs* (as well as the standard library directories) for the library *libgx.a*:

```
% icc -nx myprog.c -lnews -L../mylibs -lgx
```

- If you specify more than one **-L** switch, the named directories are searched in reverse order (the directory specified by the first **-L** switch on the command line is searched after the directory specified by the second **-L** switch on the command line). For example:

```
% icc -nx myprog.c -lnews -L../mylibs -lgx -Llocallibs -llocal
```

  This command searches for libraries as follows:

  - It searches only the standard library directories for the library *libnews.a.*
  - It searches the directory *../mylibs* and then the standard library directories for the library *libgx.a.*
  - It searches the directory *locallibs*, then *../mylibs*, and then the standard library directories for the library *liblocal.a.*

  Note that the **-L** switch also affects system libraries; in fact, directories specified by **-L** are searched for system libraries *before* the standard library directories.

Running Applications

Once you have compiled your application into an executable file (and, if necessary, copied the executable to an Paragon supercomputer), you run it by typing its name at your shell command prompt, as you would for any other compiled program.

For example, if *myapp* is a compiled application, you can execute it with the following command:

```
% myapp
```

The way the application runs depends on how you linked it and on what system calls it makes:

- If *myapp* was linked with the **-nx** switch, this command runs *myapp* on your default number of nodes in your default partition. The section "Controlling the Application's Execution Characteristics" on page 2-12 tells you more about the default partition, and about the environment variables and command-line switches you can use to control the execution characteristics of applications linked with the **-nx** switch.

- If *myapp* was linked with the **-lnx** switch, this command runs *myapp* on the nodes and partition specified by system calls within the application. The section "Managing Applications" on page 4-2 tells you how to use these system calls. If *myapp* does not specify the nodes and partition in these calls, it defaults to running on your default number of nodes in your default partition. If *myapp* does not make any of these calls, it runs on one node in the service partition.

- If *myapp* was linked without the **-nx** or **-lnx** switch, it is an ordinary non-parallel program, and it runs on one node in the service partition.

If you see the error message "request overlaps with nodes in use," it means that your default partition does not allow overlapping applications and someone else is already running an application in that partition. Try again later, or use a different partition (as described under "Running an Application in a Particular Partition" on page 2-23). You can use the **pspart** command to determine which partitions have applications running in them, as described under "Listing the Applications in a Partition" on page 2-64. You can also use the command **showpart -f** to determine which nodes in a given partition do not have applications running on them, as described under "Showing Free Nodes" on page 2-55.

If you see the error message "partition permission denied" or "exceeds partition resources," check to be sure the environment variables *NX_DFLT_PART* and *NX_DFLT_SIZE* are properly defined. See "Using the Default Partition" on page 2-13 and "Specifying Application Size" on page 2-15 for more information on these variables; see your system administrator for information on the proper settings for these variables at your site.

If you see the error message "error 216 occurred, unknown," it means that the application was compiled on a previous release of the operating system and uses an out-of-date version of the libraries. (Error 216 is "parallel application incompatible with OS release", but the "unknown" message may appear if the application is so out-of-date that it doesn't know about the existence of this error.) If this occurs, recompile the application and try again.

I/O Redirection

You can redirect the standard input, standard output, and standard error of an application with the usual OSF/1 techniques. For example, the following command redirects the input and output of the application *myapp*:

```
% myapp < myfile.in > myfile.out
```

This command runs the application *myapp* with its standard input redirected from the file *myfile.in* and its standard output redirected to the file *myfile.out.*

Note that, by default, all the nodes read and write their standard input, standard output, and standard error using PFS I/O mode 0. In mode 0, all file access requests are honored on a first-come, first-served basis. You can change this behavior by selecting a different I/O mode; see "Using I/O Modes" on page 5-14 for more information. The standard input, standard output, and standard error are line-buffered by default. This means that if all the nodes write to standard output or standard error, the output from all the nodes is intermixed in the output, line by line; if all the nodes read from standard input, each line of the input goes to an arbitrary node.

Running SMP Programs

If you are running programs that take advantage of the symmetric multiprocessing (SMP) features of the system, you may have to set the environment variable *DFLT_NCPUS* before you run your program. For more information about *DFLT_NCPUS* and the SMP programming model, see "Setting DFLT_NCPUS" on page 6-5.

Controlling the Application's Execution Characteristics

Command Synopsis	Description
application [**-sz** *size* | **-sz** *h***X***w* | **-nd** *h***X***w***:***n*] [**-rlx**] [**-pri** *priority*] [**-pt** *ptype*] [**-on** *nodespec*] [**-pn** *partition*] [**-nt** *nodetype*] [*msg_switches*] [**\;** *app2* [**-pt** *ptype*] [**-on** *nodespec*]] ...	Execute an application.

When you run an application, you can use command-line switches and environment variables to control the way the application executes. This section discusses all the switches and environment variables except for the *msg_switches*, which are used for message-passing performance tuning; for information on the *msg_switches*, see "Message-Passing Configuration Switches" on page 8-18.

Command-line switches can appear in any order on the command line, and may be intermixed with application-specific switches and arguments. If you specify the same command-line switch more than once in a single command, the last occurrence overrides the earlier ones. For example, the following two commands are equivalent:

```
% myapp -sz 4 -sz 50 -pri 8 file.dat
% myapp -pri 8 -sz 4 file.dat -sz 50
```

Each of these commands runs the application *myapp*, with the argument *file.dat*, at priority 8 on 50 nodes of your default partition.

If the application was linked with the **-nx** switch, the command-line switches discussed in this section are interpreted and removed from the command line before the application starts up. In the previous examples, the arguments **-pri 8**, **-sz 4**, and **-sz 50** are interpreted and removed by the **-nx** code; *myapp* sees only the argument **file.dat** (if *myapp* is a C program *argc* is 2, *argv[0]* is "myapp", and *argv[1]* is "file.dat").

NOTE

All the examples in this section assume that *myapp* was linked with the **-nx** switch.

An application that is not linked with **-nx** controls its own execution with system calls, as discussed under "Managing Applications" on page 4-2. Such an application may or may not obey the command-line switches discussed in this section, depending on how it was programmed.

Using the Default Partition

When you run a parallel application on the Paragon supercomputer, it runs in a *partition.* The partition determines the maximum number of nodes used by the application and how the application is scheduled, as described later in this chapter. An application stays in the same partition for its entire run.

If you do not specify otherwise, the application runs in the partition specified by the environment variable *NX_DFLT_PART.* If the environment variable *NX_DFLT_PART* is not set, the application runs in the *compute partition*, a special partition that is present on all Paragon supercomputers. The partition specified by *NX_DFLT_PART* (or, if this variable is not set, the compute partition) is called your *default partition.*

For example, to run the application *myapp* in your default partition, use the following command:

```
% myapp
```

This command runs the application *myapp* in the partition specified by the environment variable *NX_DFLT_PART*, or in the compute partition if *NX_DFLT_PART* is not set.

If you see an error message such as "partition not found" or "partition permission denied," ask your system administrator what your default partition should be, then use the commands described in the next section to set the variable *NX_DFLT_PART* to that value. You can also use the **-pn** switch (described under "Running an Application in a Particular Partition" on page 2-23) to run an application in a different partition.

For more information about partitions, see "Managing Partitions" on page 2-30.

Setting Your Default Partition

The command you use to set or change your default partition depends on which shell you use.

- If you use the C shell, use the **setenv** command. For example, if you are a C shell user, the following command sets your default partition to *mypart*:

  ```
  % setenv NX_DFLT_PART mypart
  ```

  setenv is a built-in command of the shell; see **csh** in the *OSF/1 Command Reference* for more information.

  You can put this command in your *.login* or *.cshrc* file on the Paragon supercomputer to have your default partition set to *mypart* each time you log in.

- If you use the Bourne or Korn shell, set the variable and use the **export** command to make its value available to commands other than the shell. For example, if you are a Bourne or Korn shell user, the following commands set your default partition to *mypart*:

  ```
  $ NX_DFLT_PART=mypart
  $ export NX_DFLT_PART
  ```

  You do not have to use the **export** command each time you set the variable. You only have to export a variable once in each login session. **export** is a built-in command of the shell; see **sh** or **ksh** in the *OSF/1 Command Reference* for more information.

  You can put these commands in your *.profile* file on the Paragon supercomputer to have your default partition set to *mypart* each time you log in.

You can use an absolute or relative partition pathname as the value of *NX_DFLT_PART*. For example, the following C shell commands are equivalent:

```
% setenv NX_DFLT_PART myorg.mypart
% setenv NX_DFLT_PART .compute.myorg.mypart
```

See "Partition Pathnames" on page 2-33 for more information on partition pathnames.

If you use the C or Korn shell, you can create an alias to change your default partition. For example, the following C shell command creates a "setpart" alias that sets your default partition to its argument:

```
    % alias setpart 'setenv NX_DFLT_PART \!*'
```

Determining the Current Default Partition

To find out your default partition once you have set it, use the **echo** command. For example:

```
% echo $NX_DFLT_PART
mypart
```

This command works the same in any shell.

Specifying Application Size

An application's *size* is the number of nodes allocated to the application from the partition. The processes of the application run only on this set of nodes, and do not exchange messages with processes on nodes outside this set. Depending on the characteristics of the partition, this allocation may or may not be exclusive: some or all of these nodes may also be allocated to other applications and/or other partitions. An application keeps the same size for its entire run.

To set an application's size, use the switch **-sz** *size*, where *size* is any positive integer less than or equal to the number of nodes in the partition. For example, to run the application *myapp* on 64 nodes of your default partition, use the following command:

```
% myapp -sz 64
```

The **-sz** *size* switch attempts to allocate a square group of nodes if it can. If this is not possible, it attempts to allocate a rectangular group of nodes that is either twice as wide as it is high or twice as high as it is wide. If this is not possible, it allocates any available nodes; in this case, nodes allocated to the application may not be *contiguous* (that is, they may not all be physically next to each other). If the requested number of nodes is not available, the command fails and the application does not run; an error message is printed to explain why the specified number of nodes is not available.

No matter what the shape of the application, node numbers within the application (as returned by **mynode**()) will always be sequential from 0.

Specifying a Rectangle of Nodes

To force allocation of a contiguous rectangle of a particular size and shape, use the switch **-sz** *h***X***w*, where *h* and *w* are positive integers that specify the height and width of the desired rectangle. (You can use an uppercase or lowercase letter **X** between the integers *h* and *w*.) For example, to run *myapp* on an 8 by 8 node rectangle of your default partition, use the following command:

```
% myapp -sz 8x8
```

If successful, this command runs *myapp* on an 8 by 8 node rectangle of nodes, which could be located anywhere within the partition that it fits. If no 8 by 8 node rectangle is available in the default partition, the command fails immediately and the application does not run, even if there are 64 nodes free in the partition. If this occurs, the command fails with the error message "exceeds partition resources" if no such rectangle can be found that fits within the partition, or "request overlaps with nodes in use" if the rectangle fits within the partition but some of its nodes are busy).

Specifying a Particular Rectangle of Nodes

To force allocation of a contiguous rectangle of a particular size and shape at a particular location within the partition, use the switch **-nd** *h***X***w***:***n*. (This switch is called **-nd**, rather than **-sz**, because it specifies a particular set of nodes rather than just a size or shape.)

In the **-nd** *h***X***w***:***n* switch, *h* and *w* are positive integers that specify the height and width of the desired rectangle, and *n* is a positive integer that specifies the node number within the partition for the upper left corner of that rectangle. You can use an uppercase or lowercase letter **X** between the integers *h* and *w*. When choosing the value of *n*, remember that in an *m*-node partition the nodes are numbered left to right and top to bottom from 0 to *m*–1.

For example, to run *myapp* on an 8 by 8 node rectangle in the upper left corner of your default partition, use the following command:

```
% myapp -nd 8x8:0
```

In this case, if the specified nodes are not available in the default partition, the application fails immediately (even if there is a different 8 by 8 node rectangle available).

Using the Default Size

If you don't use the **-sz** or **-nd** switch, the application's size is specified by the environment variable *NX_DFLT_SIZE*, whose value must be a single positive integer. You can use the techniques discussed for the *NX_DFLT_PART* variable in the previous section to get and set the value of the *NX_DFLT_SIZE* variable. If *NX_DFLT_SIZE* is not set, the application runs on all available nodes of the partition, and its size is set to the size of the partition. The size specified by *NX_DFLT_SIZE* (or, if this variable is not set, the size of the partition) is called your *default number of nodes*.

An application can determine its size by calling **numnodes()**, and each process in the application can determine its node number within the application by calling **mynode()**. **mynode()** returns a node number from 0 to one less than the application's size. (See "Process Characteristics" on page 3-3 for more information on these calls.) For example, with **-sz 64**, **-sz 8x8**, or **-nd 8x8:0**, **numnodes()** returns 64 and **mynode()** returns a number from 0 to 63 inclusive. There is no way for an application to change its size.

An application can determine its shape by calling **nx_app_rect()**, which returns the height and width of the rectangle of nodes allocated to the application. If the nodes allocated to the application do not form a rectangle, **nx_app_rect()** returns a height of 1 and a width equal to **numnodes()**. (**nx_app_rect()** can also be called by the name **mypart()** for compatibility with the Touchstone DELTA System.)

Relaxing Application Size

No matter how you specify the application size, if any of the nodes you request are not available, the application fails with an error message and does not run. The "availability" of a node is determined by the partition's scheduling type and whether or not the node is already in use; for example, if the partition does not permit overlapping applications, any node that already has an application running on it cannot be allocated. See "Scheduling Characteristics" on page 2-39 for more information. A node can also be unavailable due to a software or hardware problem; see "Unusable Nodes" on page 2-37.

You can use the switch **-rlx** to relax the requirement that the exact specified number of nodes must be available. When you use **-rlx**, the application may run on *fewer* nodes than you requested. In other words, the application runs on *as many nodes as possible*, *up to* the requested number of nodes. However, there must be at least one node available or the command still fails.

NOTE

> **-rlx** can be used to relax the default size, the **-sz** *size* switch, or the **-nd** switch. It cannot be used together with the switch **-sz** *h***X***w*.

For example, if the environment variable *NX_DFLT_SIZE* is not set, the following command runs the application *myapp* on every available node in the default partition:

```
% myapp -rlx
```

The following command runs the application *myapp* on *up to* ten nodes of the default partition. If less than ten nodes are available, the application runs on all the available nodes:

```
% setenv NX_DFLT_SIZE 10
% myapp -rlx
```

The following command runs the application *myapp* on *up to* five nodes of the default partition. If less than five nodes are available, the application runs on all the available nodes:

```
% myapp -sz 5 -rlx
```

In any of the above cases, if *no* nodes are available in the default partition, the command fails.

The following command runs the application *myapp* on *up to* a 3-by-3-node rectangle of nodes located in the upper left corner of the default partition. If any of those nodes is not available, the application runs on the remaining nodes of that rectangle.

```
% myapp -nd 3x3:0 -rlx
```

In this case, if no nodes are available *in the specified rectangle*, the command fails.

Specifying Application Priority

An application's *priority* is an integer associated with the application that is used in determining how much of a node's processor time the application gets when the node is allocated to more than one application at once. 0 is the lowest priority, and 10 is the highest.

The application's priority is only one of several factors that determine how much processor time it gets. For example, the application's processor time can be affected by the priorities of other applications in the system and by the *effective priority limit* of the partition in which the application runs. See "Scheduling Characteristics" on page 2-39 for more information.

To set the priority of the application, use the switch **-pri** *priority*, where *priority* is an integer from 0 to 10 inclusive. If you don't use the **-pri** switch, the application's priority is set to 5.

For example, to run the application *myapp* with a priority of 6, use the following command:

```
% myapp -pri 6
```

An application can change its priority by calling **nx_pri()** (see "Setting an Application's Priority with nx_pri()" on page 4-15 for more information).

Specifying Process Type

A process's *process type*, or *ptype*, is an integer associated with the process that differentiates it from any other process in the application that is on the same node. The process's node number and process type together form the process's "address" for messages within the application.

To set the process type of each process in the application, use the switch **-pt** *ptype*, where *ptype* is an integer from 0 to 2,147,483,647 ($2^{31} - 1$) inclusive. If you don't use the **-pt** switch, the process type of each process is 0.

For example, to run the application *myapp* with a process type of 1 for each process, use the following command:

```
% myapp -pt 1
```

A process can find out its current process type by calling **myptype()**. For example, with ***-pt 1***, **myptype()** returns 1 on all nodes. Once a process's process type has been set to a valid value, it cannot change its process type and no other process in the same application on the same node can use that process type for the run of the application. See "Process Characteristics" on page 3-3 for information on process types and the **myptype()** and **setptype()** system calls.

The **-pt** switch is most commonly used when running multiple programs in one application, as discussed under "Running Applications Consisting of Multiple Programs" on page 2-21. In most other circumstances, you can use the default process type of 0.

Running a Program on a Subset of the Nodes

Usually you run the same program file on all the nodes allocated to the application from the partition. However, you can also run a program on just some of the nodes, leaving the other nodes vacant for other programs. When you do this, the other nodes are allocated to the application, but no processes are started on them.

To run a program on a subset of the nodes of an application, use the switch **-on** *nodespec*, where *nodespec* is one of the following:

x The node whose node number is *x*.

x..*y* The range of nodes from numbers *x* to *y*.

n The last node of the partition.

nspec[,*nspec*]... The specified list of nodes, where each *nspec* is a node specifier of the form *x*, *x*..*y*, or **n** (no node may appear more than once in this list). Do not put any spaces in this list.

If you don't use the **-on** switch, the program is run on all nodes allocated to the application.

NOTE

The numbers you use with **-on** are node numbers within the application (which always range from 0 to one less than the size of the application), not node numbers within the partition.

For example, to run the program *myapp* on the first three nodes of a 20-node application, use the following command:

```
% myapp -sz 20 -on 0,1,2
```

This command creates an application of size 20 in your default partition and runs *myapp* on nodes 0, 1, and 2 of the application. Within this application, the function **numnodes()** returns 20, and the function **mynode()** returns a number from 0 to 19 inclusive. However, no processes are started on nodes 3 through 19.

You can use the letter **n** to represent "the last node in the application." For example, the following command creates an application of your default size in your default partition and runs *myapp* on the first and last nodes of the application:

```
% myapp -on 0,n
```

For example, if your *NX_DFLT_SIZE* variable is set to 64 (and there are at least 64 nodes in your default partition), this would run *myapp* on nodes 0 and 63 of the application.

You can also use a pair of numbers separated by two periods (*x..y*) to specify "nodes *x* through *y* inclusive." For example, the following command creates an application of size 100 in your default partition and runs the program *myapp* on nodes 10 through 90:

```
% myapp -sz 100 -on 10..90
```

It doesn't matter whether *y* is greater than *x* or vice versa. For example, the following command *also* creates an application of size 100 in your default partition and runs the program *myapp* on nodes 10 through 90:

```
% myapp -sz 100 -on 90..10
```

These notations can be combined. For example, the following command creates an application of your default size in your default partition and runs *myapp* on all nodes but node 0 of the application:

```
% myapp -on 1..n
```

Another example: the following command creates an application of your default size in your default partition and runs *myapp* on node 1, node 3, nodes 5 through 10 inclusive, and the last node of the application:

```
% myapp -on 1,3,5..10,n
```

NOTE

Do not use **-on** if you just want to run a single program on a specific number of nodes.

The **-on** switch is designed to be used when running multiple programs as a single application, as discussed in the next section. You can also use the **-on** switch to run a "manager" program on one or a few nodes of an application; the "manager" program can then run "worker" programs on other nodes by calling **nx_nfork()**, **nx_load()**, or **nx_loadve()** (see "Managing Applications" on page 4-2 for information on these functions).

The **-on** switch is *not* designed to run an application on a particular number of nodes or a particular set of nodes. If you want to run an application on a particular number of nodes, use the **-sz** switch; if you want to run an application on a particular set of nodes, use the **-nd** switch.

If you use **-on** when you should be using **-sz** or **-nd**, the application will be allocated more nodes than it needs. Also, if you use **-on** and do not run a program on every node of the application, global synchronizing operations will hang. (Global synchronizing operations, such as **gdsum()** and **gopen()**, block until they are called by every node in the application. If you run a program on only a subset of the nodes, these operations will block forever. See "Global Operations" on page 3-27 and "Synchronization Summary" on page 5-54 for information on global synchronizing operations.)

Running Applications Consisting of Multiple Programs

You can run multiple program files as a single application. For example, you could run two or more separate programs on every node (the resulting processes must have different process types, and the processes time-share the processor while the application is active). You might also run a manager program on one node and worker programs on the other nodes. The programs should be written to work together; you would not usually run two arbitrary programs together in one application.

To run multiple program files as a single application, use the following syntax:

```
% file [ switches ] [ \; file [ -pt ptype ] [ -on nodespec ] ] ...
```

That is, you use two or more complete commands on one line, separated by an escaped semicolon (backslash followed by semicolon).

NOTE

> The escaped semicolon (\;) must be preceded and followed by a space or tab. Otherwise, it will be considered part of the preceding or following argument.

The first *file* must either have been linked with **-nx** or must call **nx_initve...()** without overriding the command line; the second and subsequent *files* may have been linked with or without **-nx**, but must *not* call **nx_initve...()**.

The command-line switches you can use with the *file*s are different:

- You can use any application switches (**-sz**, **-nd**, **-rlx**, **-pri**, **-pt**, **-on**, **-pn**, **-nt**, and *msg_switches*) with the first file. (The **-pn** and **-nt** switches are discussed later in this chapter; see "Message-Passing Configuration Switches" on page 8-18 for information on the *msg_switches*). The effect of these switches varies according to the switch:
  - The **-sz**, **-nd**, **-rlx**, **-pri**, **-pn**, **-nt**, and *msg_switches* switches you use with the first file affect the entire application.
  - The **-pt** and **-on** switches you use with the first file affect the first file only.
- If you want **-pt** and **-on** to affect second and subsequent files, you must specify them again for each file. These switches affect the associated file only.

If you run multiple processes on a single node, you must use the **-pt** switch to specify a unique process type for each process. When two or more processes in an application run on the same node, each must have a different process type. If you don't use the **-pt** switch, each process will have process type 0, and you will receive an error message.

For example, to run the programs *myapp* and *myapp2* as a single application, use the following command:

```
% myapp \; myapp2 -pt 1
```

This command runs the program *myapp* with process type 0 and the program *myapp2* with process type 1 on your default number of nodes in your default partition.

To run the program *manager* on node 0 of a 20-node application and the program *worker* on the remaining nodes, use the following command:

```
% manager -sz 20 -on 0 \; worker -on 1..n
```

Note that **-on** is specified twice, once for each file. This command creates an application of size 20 in your default partition. It then runs the program *manager* on node 0 of the application and the program *worker* on nodes 1 through 19 of the application. All the resulting processes have process type 0, but this does not create a conflict because *manager* and *worker* run on different nodes.

NOTE

If you forget the backslash before the semicolon, the first program is run as an application by itself and the second program runs after the first program finishes. This usually results in unexpected behavior from the programs.

Running an Application in a Particular Partition

To run an application in a partition other than your default partition, use the switch **-pn** *partition*. You must have execute permission for the specified partition. The partition specified by **-pn** overrides the value of *NX_DFLT_PART*, if any. If you don't use the **-pn** switch, the application runs in your default partition, as described under "Using the Default Partition" on page 2-13.

NOTE

If your default number of nodes, as specified by the environment variable *NX_DFLT_SIZE*, is greater than the number of nodes available in the specified partition, you may get a "partition resources exceeded" or "request overlaps with nodes in use" error.

If you see this error, use the **-sz** switch or change the value of *NX_DFLT_SIZE* to specify an application size less than or equal to the size of the specified partition.

For example, to run the application *myapp* on your default number of nodes in the partition *mypart*, use the following command:

```
% myapp -pn mypart
```

You can use an absolute or relative partition pathname with **-pn** (see "Partition Pathnames" on page 2-33 for information on partition pathnames). For example, the following commands are equivalent:

```
% myapp -pn myorg.mypart
% myapp -pn .compute.myorg.mypart
```

For more information about partitions, see "Managing Partitions" on page 2-30.

Running an Application on a Particular Node Type

On some Paragon systems, not all the nodes in the compute partition have the same hardware. For example, some nodes may have more memory than others, or some nodes may have I/O interfaces that the others do not. The hardware characteristics of each node are described by a comma-separated series of strings called *attributes*. You can use the command **showpart -l** or **lspart -l** to see the attributes of the nodes in a partition, as discussed under "Showing Partition Characteristics" on page 2-54 and "Listing Subpartitions" on page 2-60.

Node Attributes

The meanings of the most commonly-seen node attributes are shown in Table 2-1. Other node attributes (such as additional node or I/O types) may also be present on your system. Attributes are not case-sensitive; for example, **GP**, **gp**, and **Gp** are all equivalent.

Table 2-1. Node Attributes

Attribute	Meaning
bootnode	Boot node.
gp	GP (two-processor) node.
mp	MP (three-processor) node. This includes MP-as-GP nodes.
mcp	Node with a message coprocessor.
*n***proc**	Node with *n* application processors (not counting the message coprocessor).
*n***mb**	Node with *n*M bytes of physical RAM.
io	Any I/O node.
net	I/O node with any type of network interface.
enet	Network node with Ethernet interface.
hippi	Network node with HIPPI interface.
scsi	I/O node with a SCSI interface.
disk	SCSI node with any type of disk.
raid	Disk node with a RAID array.
tape	SCSI node with any type of tape drive.
3480	Tape node with 3480 tape drive.
dat	Tape node with DAT drive.
IDstring	SCSI node whose attached device returned the specified *IDstring* (supplied by the device manufacturer) at boot time. For example, a disk node might have the *IDstring* **NCR ADP-92/01 0304**.

NOTE

In the current release, the only supported configuration is for one processor on each node to be a message coprocessor. This means that all nodes are **mcp** nodes, GP and MP-as-GP nodes are **1proc**, and MP nodes are **2proc**.

An attribute that is indented in the first column of Table 2-1 is a more specific version of the attribute at the previous level of indentation. For example, **net** and **scsi** nodes are specific types of **io** node; **enet** and **hippi** nodes are specific types of **net** node (and thus also specific types of **io** node).

When each node boots, the operating system gets the node's attributes and stores them as a string. This string includes *all* the attributes associated with the node (both the more-specific and less-specific versions). When you request a node with a particular attribute, the system searches these strings and returns only nodes whose strings contain the specified attribute. This means that requesting a less-specific attribute may match a node with any of the more-specific types indented below it. For example, a request for a **scsi** node may allocate a **disk** node or a **tape** node; however, it will never allocate any type of **net** node (unless that node is a **scsi** node as well).

For example, a GP disk node with 32M bytes of memory might have the following attributes string:

```
GP,mcp,1proc,32mb,io,scsi,disk,NCR ADP-92/01 0302
```

- If you request a **gp** node, an **io** node, a **scsi** node, or a **disk** node, you might get the above node, because all these attributes appear in the string.

- If you request an **mp** node, a **net** node, an **enet** node, or a **tape** node, you will *not* get the above node, because none of these attributes appear in the string.

Specifying Node Attributes

Sometimes you might want to run an application only on nodes that have a certain attribute or set of attributes; for example, only those nodes with 16M bytes or more of RAM. To do this, use the switch **-nt** *nodetype*, where *nodetype* is one of the following:

attribute Selects nodes having the specified attribute. The standard node attributes are shown in Table 2-1. For example, the string **mp** selects only MP nodes.

!*attribute* Selects nodes *not* having the specified attribute. No white space may appear between the **!** and the *attribute*. For example, the string **!io** selects only nodes that are *not* I/O nodes.

[*relop*][*value*]*attribute*

Selects nodes having a specified value or range of values for the attribute:

- The *relop* can be =, >, >=, <, <=, !=, or ! (!= and ! mean the same thing). If the *relop* is omitted, it defaults to =.

- The *value* can be any nonnegative integer. If the *value* is omitted, it defaults to 1.

- The *attribute* can be any attribute shown in Table 2-1, but is usually either **proc** or **mb**. (Other attributes have the value 1 if present or 0 if absent.)

No white space may appear between the *relop*, *value*, and *attribute*.

For example, the string **>=16mb** selects nodes with 16M bytes or more of RAM; **32mb** selects nodes with exactly 32M bytes of RAM; **>proc** selects nodes with more than one processor.

ntype[,*ntype*]...

Selects nodes having *all* the attributes specified by the list of *ntype*s, where each *ntype* is a node type specifier of the form *attribute*, **!***attribute*, or [*relop*][*value*]*attribute*. You can use white space (space, tab, or newline) on either side of each comma, but not within an *ntype*.

For example, the string **mp,32mb** selects MP nodes with exactly 32M bytes of RAM; **io,gp,>16mb** selects GP-based I/O nodes with more than 16M bytes of RAM; **io,!enet** selects I/O nodes that are *not* Ethernet nodes.

Quoting Node Attributes

If any characters that are special to your shell (such as **>**, **<**, or white space) appear in a *nodetype* string, you must enclose the entire *nodetype* in quotes or precede each special character with a backslash. For example:

```
% myapp -nt "mp, >16mb"
```

If you use the C shell, the special character **!** must *always* be preceded by a backslash, even if the *nodetype* is quoted. For example:

```
% myapp -nt \!gp
```

Using Node Attributes with No Application Size

If you use the **-nt** switch and do not specify an application size (that is, if you don't use the **-sz** switch, the **-nd** switch, or the environment variable *NX_DFLT_SIZE*), the application runs on all the nodes in the partition that have the specified attributes. If any of the specified nodes is not available (for example, if the partition does not allow overlapping applications and one or more nodes of the specified type already has an application running on it), the command fails with an error message and the application does not run.

For example, the following command runs the application *myapp* on all the MP nodes in the default partition:

```
% myapp -nt mp
```

The following command runs the application *myapp* on all the GP nodes in the default partition that are not I/O nodes (note that the exclamation point is escaped):

```
% myapp -nt gp,\!io
```

The following command runs the application *myapp* on all the nodes in the default partition that have one processor and more than 16M bytes of memory (note that the *nodetype* is quoted, because it contains a space and the special character >):

```
% myapp -nt "1proc, >16mb"
```

The above examples assume that the environment variable *NX_DFLT_SIZE* is not set.

Using Node Attributes with an Application Size

If you use the **-nt** switch together with the **-sz** switch, the **-nd** switch, or the environment variable *NX_DFLT_SIZE*, the application runs on the specified nodes with the specified attributes, as follows:

- For **-sz** *size* or *NX_DFLT_SIZE*, at least the specified number of nodes with the specified attributes must be available in the partition.
- For **-sz** *h***X***w*, at least one rectangle of nodes of the specified size and shape, all of which have the specified attributes, must be available somewhere in the partition.
- For **-nd** *h***X***w***:***n*, the specified rectangle of nodes must be available and all the nodes must have the specified attributes.

If the specified nodes with the specified attributes are not available in the partition, the command fails with an error message and the application does not run.

For example, the following command runs the application *myapp* on 5 MP nodes in the default partition (it fails if less than 5 MP nodes are available):

```
% myapp -sz 5 -nt mp
```

The following command also runs the application *myapp* on 5 MP nodes in the default partition (it fails if less than 5 MP nodes are available):

```
% setenv NX_DFLT_SIZE 5
% myapp -nt mp
```

The following command runs the application *myapp* on a 2-by-4-node rectangle of MP nodes in the default partition (it fails if no such rectangle of MP nodes is available anywhere in the partition):

```
% myapp -sz 2x4 -nt mp
```

The following command runs the application *myapp* on a 3-by-3-node rectangle of MP nodes in the upper left corner of the default partition (it fails if the specified rectangle is not available or does not consist entirely of MP nodes):

```
% myapp -nd 3x3:0 -nt mp
```

Using Node Attributes with a Relaxed Application Size

As discussed under "Relaxing Application Size" on page 2-17, you can use the **-rlx** switch to relax the requirement that a specified number of nodes must be available. When you use **-rlx** together with **-nt**, the application still runs only on nodes of the type you specify, but it may run on *fewer* nodes than you requested. In other words, the application runs on *as many nodes as possible* having the specified type, *up to* the requested number of nodes. However, there must be at least one node of the specified type available or the command fails.

For example, if the environment variable *NX_DFLT_SIZE* is not set, the following command runs the application *myapp* on every available MP node in the default partition:

```
% myapp -nt mp -rlx
```

Without the **-rlx** switch, this command would fail if any of the MP nodes in the default partition was not available (for example, if the partition did not allow overlapping applications and one or more of the MP nodes already had an application running on it). With the **-rlx** switch, this command runs the application on as many MP nodes as it can get.

Another example: the following command runs the application *myapp* on *up to* ten MP nodes (if less than ten MP nodes are available in the default partition, the application runs on all the available MP nodes):

```
% setenv NX_DFLT_SIZE 10
% myapp -nt mp -rlx
```

The following command runs the application *myapp* on *up to* five MP nodes (if less than five MP nodes are available in the default partition, the application runs on all the available MP nodes):

```
% myapp -nt mp -sz 5 -rlx
```

In any of the above cases, if *no* MP nodes are available in the default partition, the command fails.

The following command runs the application *myapp* on *up to* a 3-by-3-node rectangle of MP nodes located in the upper left corner of the default partition (if any of those nodes is not available or is not an MP node, the application runs on the remaining nodes of that rectangle):

```
% myapp -nt mp -nd 3x3:0 -rlx
```

In this case, if no MP nodes are available *in the specified rectangle*, the command fails.

Managing Running Applications

You use the standard OSF/1 techniques to manage running applications. For example, you use your interrupt key (usually **<Del>** or **<Ctrl-c>**) to interrupt a running application. If you use the C shell or Korn shell, you can use your suspend key (usually **<Ctrl-z>**) to suspend an application, and the **fg** or **bg** command to resume it. See **csh**, **sh**, or **ksh** in the *OSF/1 Command Reference* for more information on these techniques.

NOTE

Interrupting or suspending an application that is "rolled out" will not take effect until the application is "rolled in" again.

Parallel applications can be *gang-scheduled* to make more efficient use of system resources. In gang scheduling, an application is allowed to run for a time period, called the *rollin quantum*, and then is "rolled out" and another application is "rolled in" in its place. If the rollin quantum is long, you may not see any response to a **<Ctrl-c>** or **<Ctrl-z>** for a long time. See "Scheduling Characteristics" on page 2-39 for more information on gang scheduling.

You can also use the **ps** command to determine the status of an application, and the **kill** command to terminate it. For example:

```
% myapp &
[1] 7045
% ps
  PID TT  STAT      TIME COMMAND
 5841 p3  S  +   0:02.50 -csh (csh)
 7045 p3  R      0:00.30 myapp
% kill 7045
% ps
  PID TT  STAT      TIME COMMAND
 5841 p3  S  +   0:02.55 -csh (csh)
[1]   + Terminated          myapp
%
```

The **ps** command shows only processes running in the service partition. See **ps** and **kill** in the *OSF/1 Command Reference* for more information on these commands. To show processes running in partitions other than the service partition, use the **pspart** command.

The *myapp* process that you see in the output of **ps** is a special process called the *controlling process* that runs in the service partition; you do not see the other application processes in the output of **ps**. However, sending a signal to the controlling process with **<Del>**, **<Ctrl-c>**, **<Ctrl-z>**, or **kill** signals all the processes in the application. See "Managing Applications" on page 4-2 for more information on the controlling process.

If the application was started from the Bourne shell (**sh**) or from a shell script, you will see *two* processes with the name of the application in the output of **ps**. One of these two processes is the controlling process; the other is another special process, called the *shepherd process*. The shepherd process is necessary for the application; do not kill it. When the application terminates, this process will terminate as well.

To determine which process is which, use the command **ps -f** and examine the **PPID** (parent PID) fields of the two processes. The shepherd process is the parent of the controlling process. For example:

```
$ ps -f
USER        PID   PPID %CPU STARTED  TT        TIME COMMAND
chris    131125 131124  0.0 13:55:51 p2     0:00.28 -sh (sh)
chris    131129 131125  0.0 13:56:36 p2     0:00.05 myapp
chris    131130 131129  0.0 13:56:36 p2     0:00.03 myapp
```

In this case the second *myapp* process (PID 131130) is the controlling process. The first *myapp* process, PID 131129, is the parent of the controlling process and is therefore the shepherd process.

You can use the **pspart** command to determine the status of all the applications in a particular partition. See "Listing the Applications in a Partition" on page 2-64 for information on this command.

You can also use the Interactive Parallel Debugger (**ipd**) to control the execution of an application, down to the machine instruction. See the *Paragon™ System Interactive Parallel Debugger Reference Manual* for information on **ipd**.

Managing Partitions

The nodes of the Paragon supercomputer are divided into overlapping groups called *partitions*. When you run a parallel application, you must select a partition to run it in. The partition places limits on the execution characteristics of the application, such as which nodes it can use, whether or not it can use nodes that are already in use, and how long it can use them before it is "rolled out" and another application is "rolled in."

Depending on the policies of your site, you may or may not have to know any more about partitions than what has been discussed in this chapter so far.

- At some sites, the system administrator configures all the partitions; ordinary users can simply set the *NX_DFLT_PART* variable to an appropriate value (or leave it unset and use the compute partition) and then forget all about partitions. If your site is like this, you do not have to read this section. However, you may wish to read it to help you understand how the system works.

- At other sites, users create and configure their own partitions. If your site is like this, you should read this section.

This section includes the following information about partitions:

- Some special partitions that every Paragon supercomputer has.
- Specifying partitions with partition pathnames.
- The characteristics of a partition.
- Making partitions with the **mkpart** command.
- Removing partitions with the **rmpart** command.
- Showing the characteristics of a partition with the **showpart** command.
- Listing the subpartitions of a partition with the **lspart** command.
- Listing the applications in a partition with the **pspart** command.
- Changing the characteristics of a partition with the **chpart** command.

Special Partitions

Every Paragon supercomputer has three special partitions:

- The *root partition* directly or indirectly contains all the other partitions in the system. It is the only partition that does not have a parent partition.
- The *service partition* is the partition in which the users' shells and other commands run. Its parent is the root partition.
- The *compute partition* is the partition in which parallel applications run. Its parent is also the root partition.

The characteristics of these partitions are determined by the system administrator. In particular, the system administrator sets the ownership and permissions of these partitions according to local policies. These ownerships and permissions determine whether or not ordinary users can create partitions for their own use, or whether they must run applications in partitions provided for them by the system administrator. If ordinary users are allowed to create partitions, the system administrator can also place restrictions on the characteristics of partitions they create and the use of certain application switches within partitions.

Typically, the service partition and compute partition are the only two children of the root partition and do not overlap. However, the system administrator can choose to configure these partitions differently, and may also create additional child partitions of the root partition.

For example, some systems have an *I/O partition*: a third child of the root partition, which does not overlap with either the service or compute partitions, and which contains the nodes that control disks and other I/O devices. In other systems, the I/O "partition" is not a true partition, but a set of nodes in the root partition that are not part of either the service or the compute partition.

The Root Partition

The *root partition* is the basis for all other partitions. The name of the root partition is . (dot).

The root partition contains every usable node in the system. Depending on the underlying hardware, there may be unusable nodes within the root partition as well. The root partition organizes all the nodes in the system into a two-dimensional grid, or mesh. For example, Figure 2-1 shows the root partition of a 32-node system that is configured as a 4 by 8 node mesh. The nodes are numbered from 0 to 31.

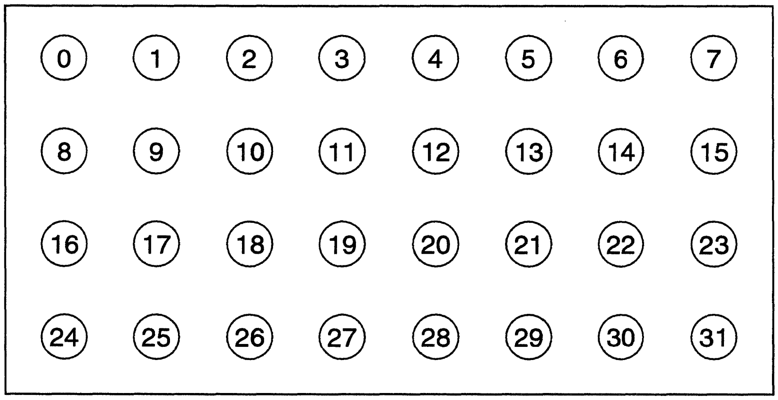


Figure 2-1. The Root Partition of a 32-Node System

NOTE

The root partition is always rectangular. (This is *not* true of partitions other than the root partition.)

For example, a system with 31 nodes would also be a 4-by-8-node rectangle, numbered as shown in Figure 2-1, but one of the nodes would be an *unusable node*, as described under "Unusable Nodes" on page 2-37. You would not be able to start any processes or allocate any subpartitions using this node.

The Service Partition

The *service partition* is the partition in which the users' shells, OSF/1 commands, and other non-parallel programs run. The name of the service partition is *service*. The service partition may not contain any subpartitions.

When you log into the Paragon supercomputer, a shell is started for you on a node in the service partition; when you execute a command in this shell, the command runs on a node in the service partition. Note that the node the command runs on is not necessarily the same node that the shell runs on; the system starts each new process on the node that is currently the least busy.

The Compute Partition

The *compute partition* is the partition in which parallel applications run. The name of the compute partition is *compute*.

When you execute a parallel application, one process (called the *controlling process*) runs in the service partition; the other processes of the application run in the compute partition, or in a subpartition of the compute partition. You can specify which partition an application runs in when you execute it.

Your system administrator determines whether or not you can create subpartitions in the compute partition and whether or not you can execute applications in the compute partition itself. There may also be other local policies that affect how you use the compute partition; for example, you may be required to run your applications in certain subpartitions during the day and others at night.

Partition Pathnames

Since partitions have a hierarchical structure like directories, they also have *pathnames* like directories. Like a file or directory pathname, a *partition pathname* identifies a partition within the hierarchical partition structure by describing the path from a known location to the specified partition.

Unlike file and directory pathnames, however, partition pathnames use a dot (.) instead of a slash (/) to separate the elements of the pathname. This is why the name of the root partition is . (dot). There is also no special partition pathname for "current partition" or "parent of the current partition." Also, you cannot use wildcards (* and ?) in partition pathnames.

There are two types of partition pathnames:

- An *absolute partition pathname* specifies the path from the root partition to the specified partition. An absolute partition pathname begins with a dot (.)

- A *relative partition pathname* specifies the path from the compute partition to the specified partition. A relative partition pathname does not begin with a dot.

NOTE

Relative partition pathnames are always relative to the compute partition (there is no "current partition").

The absolute partition pathnames of the root partition, service partition, and compute partition are . (dot)*, .service*, and *.compute* respectively. Because these partitions are not subpartitions of the compute partition, they do not have relative partition pathnames.

If the partition *mypart* is a subpartition of the compute partition, its absolute partition pathname is *.compute.mypart* and its relative partition pathname is just *mypart.*

If *subpart* is a subpartition of *mypart*, its absolute partition pathname is *.compute.mypart.subpart* and its relative partition pathname is *mypart.subpart.*

Partition Characteristics

Each partition has the following characteristics:

- A *parent partition* that contains it.

- A *name* that identifies it.

- A set of *nodes* that is allocated to it.

- An *owner* and *group* and a set of *protection modes*, like those of a file or directory, that determine what actions a given user is allowed to perform on it.

- A set of *scheduling characteristics* that determine how applications are scheduled in it.

A partition's characteristics are set when the partition is created. The **mkpart** command, described under "Making Partitions" on page 2-46, lets you specify most of these characteristics on the command line; if you don't specify otherwise, the characteristics of a new partition are set to the same values as those of its parent partition.

You can use the **showpart** command, described under "Showing Partition Characteristics" on page 2-54, to determine a partition's current characteristics.

A partition's parent partition and nodes cannot be changed. You can change the other characteristics with the **chpart** command, described under "Changing Partition Characteristics" on page 2-68.

Parent Partition

Each partition is contained within another partition. The containing partition is called the *parent* partition, and the contained partition is called a *child* partition or *subpartition* of the parent partition. (There is one exception to this rule: the root partition has no parent.)

You specify a partition's parent when you create it with **mkpart**. The parent partition determines the set of nodes that are available to be allocated to the new partition (a partition cannot include any nodes other than the nodes of its parent). The parent partition also determines the default characteristics of the new partition, as mentioned earlier. A partition's parent does not change for the life of the partition.

Partition Name

Each partition is identified by a *name*. A partition's name must be unique among all the partitions with the same parent. Partition names can be any length, but must consist of only uppercase letters (A-Z), lowercase letters (a-z), digits (0-9), and underscores (_).

You specify a partition's name when you create it with **mkpart**, and you can use **chpart** to change an existing partition's name (you must have write permission on the partition's parent partition).

Nodes Allocated to the Partition

Each partition has a set of *nodes* allocated to it from its parent partition. Depending on the characteristics of the parent partition, this allocation may or may not be exclusive: some or all of these nodes may also be allocated to other partitions and/or applications. The number of nodes in this set is called the partition's *size*.

You can specify the set of nodes allocated to the partition when you create it with **mkpart**. You can specify the partition's size and let the operating system select the nodes, or you can specify certain node numbers from the parent partition. If you don't specify either, the new partition consists of all the nodes of the parent partition.

The set of nodes allocated to a partition does not change for the life of the partition (that is, partitions never move or change their size or shape). Depending on how you allocate the nodes, they may or may not be *contiguous* (all adjacent to each other). Figure 2-2 shows examples of contiguous and noncontiguous partitions.

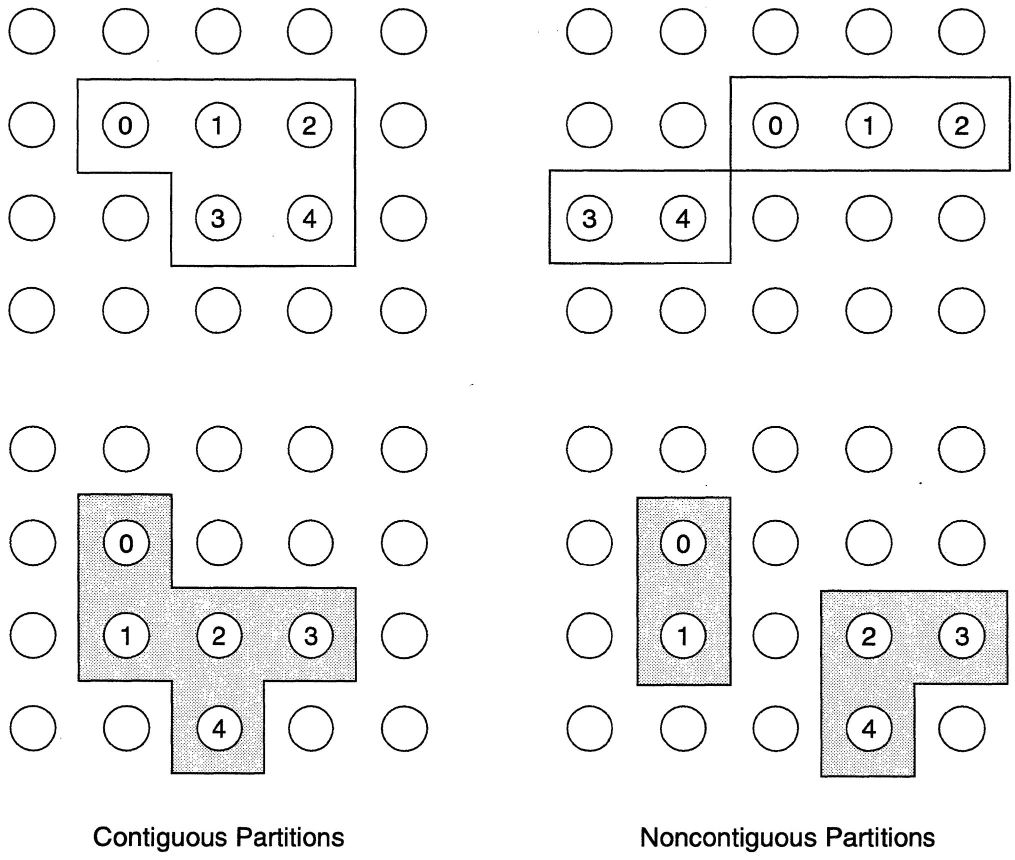


Figure 2-2. Node Numbers in Contiguous and Noncontiguous Partitions

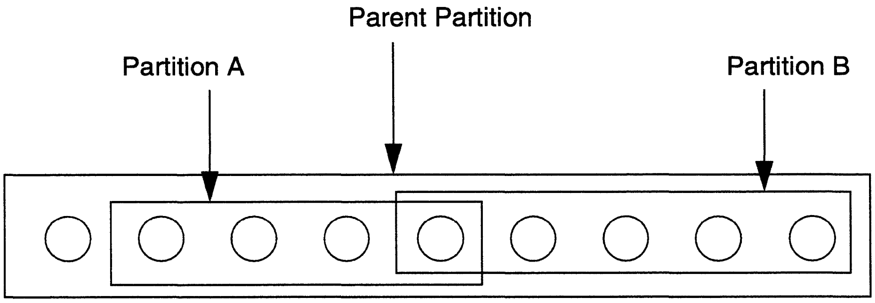
Node Numbers Within a Partition

Each node in a partition has a *node number* within the partition: an integer from 0 to one less than the partition's size. The nodes in a partition are typically numbered from left to right and then from top to bottom, as shown in Figure 2-2.

NOTE

Because partitions can overlap, a single physical node can have many logical node numbers.

For example, Figure 2-3 shows two partitions, called Partition A and Partition B, that have the same parent partition. Partition A consists of nodes 1 through 4 of the parent partition, and Partition B consists of nodes 4 through 8 of the parent partition. In this case, node 4 of the parent partition is also known as node 3 of Partition A and node 0 of Partition B.



Partition	Node Numbers								
Parent	0	1	2	3	4	5	6	7	8
A	--	0	1	2	3	--	--	--	--
B	--	--	--	--	0	1	2	3	4

Figure 2-3. Node Numbers in Overlapping Partitions

Unusable Nodes

Occasionally a node may become *unusable* because of a hardware or software failure. If this occurs, the node is still allocated to any partitions to which it was allocated before it became unusable, but no applications can be run on that node and no new partitions can include that node until the node becomes usable again. The **showpart** and **lspart** commands indicate if there are any unusable nodes in a partition.

For example, suppose you make a partition containing 20 nodes and later one of those nodes becomes unusable. If you attempt to run an application or make a subpartition with all 20 nodes of this partition while the node is unusable, the attempt will fail.

Owner, Group, and Protection Modes

Each partition has an *owner*, a *group*, and a set of *protection modes*, like those of a file or directory, that determine who can perform what operations on the partition.

When you create a partition with **mkpart**, you become the new partition's owner; the new partition's group is set to your current group (see **newgrp** in the *OSF/1 Command Reference* for more information on groups). If you are the owner of a partition, you can use **chpart** to change an existing partition's group; only the system administrator can change an existing partition's ownership.

A partition's protection modes consist of three groups of three *permission bits* (read, write, and execute for owner; read, write, and execute for group; and read, write, and execute for "other"), as described for the **chmod** command in the *OSF/1 Command Reference*. The read, write, and execute permission bits have the following meanings for a partition:

r (read)	Allows listing the subpartitions and characteristics of the partition.
w (write)	Allows creating and removing subpartitions in the partition and changing the partition's characteristics.
x (execute)	Allows executing applications in the partition.

The system administrator (*root*) is not affected by these permission bits. *root* can do anything to any partition at any time.

The permission bits can be expressed as a three-digit octal number (as for the **chmod** command) or as a string of the form `rwxrwxrwx` (as used by the **ls -l** command, where a letter represents a bit that is "on" and a dash (`-`) represents a bit that is "off"). For example, the octal number 754 is equivalent to the string `rwxr-xr--`; both grant all permissions to the owner, read and execute permissions to the group, and read permission only to all other users.

When you create a partition with **mkpart**, you can specify its protection modes. If you don't specify a partition's protection modes when you create it, they are set to the same values as those of the parent partition. If you are the owner of a partition or the system administrator, you can use **chpart** to change an existing partition's protection modes.

Scheduling Characteristics

Each partition has a set of *scheduling characteristics* that determine how the applications running in the partition are scheduled (that is, how the system arbitrates between processes when there are several processes running on a single node).

You can specify a partition's scheduling characteristics when you create it with **mkpart** and change them with **chpart**. If you don't specify a partition's scheduling characteristics when you create it, they are set to the same values as those of the parent partition.

A partition uses one of three different forms of scheduling: *standard scheduling*, *space sharing,* or *gang scheduling*.

- Partitions that use *standard scheduling* use the standard OSF/1 scheduling mechanisms. This gives good response to user input, but may result in poor performance for parallel applications (when one process in the application becomes inactive, other processes that depend on that process for information have to wait until it becomes active again).

- Partitions that use *space sharing* allow only one application per node. When you run an application in a space-shared partition, the partition checks to see if another application or partition is already using the requested nodes. If any of the nodes are in use, your application fails immediately with the error message "request overlaps with nodes in use." However, if all the specified nodes are available, your application begins running immediately and continues running, without interruption, until it completes.

- Partitions that use *gang scheduling* use a modified scheduling mechanism that makes all the processes in a parallel application active at the same time. Also, where standard scheduling swaps processes in and out frequently (typically every 100 milliseconds), gang scheduling swaps applications in and out on the basis of the partition's *rollin quantum*: a time period that can be up to 24 hours long. A long rollin quantum gives good performance for parallel applications, because the application can run for a long time without being interrupted, but may result in poor response to user input (when you give input to an application that is rolled out, the application does not respond until it is rolled in again).

Standard-scheduled partitions should be used to run interactive applications and applications that are being debugged; space-shared and gang-scheduled partitions should be used to run non-interactive (typically either computationally-intensive or I/O-intensive) applications.

The following sections give you more information about these three forms of scheduling.

Standard Scheduling

Standard scheduling is the same as the scheduling technique used on single-processor OSF/1 systems. Standard scheduling is always used in the service partition.

In a partition that uses standard scheduling, each node is scheduled like a separate computer; there is no attempt to coordinate related processes running on separate processors.

NOTE

A partition that uses standard scheduling may not contain subpartitions, and may not overlap any other partitions that use standard scheduling.

In a partition that uses standard scheduling, each process has a *priority*, a number from -20 (high priority) to 20 (low priority), that is used in determining how much processor time the process gets.

Partitions that use standard scheduling give good interactive performance for each individual process in the partition. However, there is no guarantee that related processes are active at the same time. This means that a process in a parallel application running in such a partition may find itself waiting for a message from a process that is not active, which reduces the performance of the application. To avoid this problem, you can use *gang scheduling*.

Space Sharing

Space sharing, also referred to as *tiling*, is a scheduling technique that prevents partitions and applications from overlapping. (*Overlapping* means having any physical nodes in common.) Space sharing is typically used in all partitions other than the service and compute partitions. If your system administrator has disallowed gang scheduling, space sharing is used in all partitions other than the service partition. Within a space-shared partition:

- Subpartitions may not overlap other subpartitions.
- Applications may not overlap other applications.
- Active subpartitions may not overlap applications.

An *active* subpartition is a subpartition in which one or more applications is running.

NOTE

If an application is running *anywhere* in a subpartition or any of its sub-subpartitions, even on a single node, the *entire* subpartition is considered active, and is not allowed to overlap with a running application.

If a subpartition is not active (contains no running applications), it can overlap a running application, but it cannot overlap another partition.

In a space-shared partition, any attempt to create a partition or run an application that would cause an overlap fails immediately. However, once an application is successfully started, it continues running without interruption until it completes. (Exception: if a space-shared partition overlaps with another partition, the entire partition can be interrupted by applications running in that other partition. This can only occur if the space-shared partition's parent is a gang-scheduled partition.

Space sharing is the opposite of the "time sharing" used in standard scheduling and gang scheduling. In time sharing, multiple applications can use the same nodes at the same time, but each application gets only a fraction of its nodes' processor time. In space sharing, no two applications can use a node at the same time, but each application gets 100% of its nodes' processor time.

Although space sharing allows only one *application* per node, you can have more than one *process* per node within a single application. If there are multiple processes per node within an application, standard scheduling is used to schedule these processes against each other on each node.

Partitions that use space sharing may contain subpartitions, which cannot overlap. The space-shared partition itself can overlap another partition of any type, but the advantages of space sharing may be lost if space-shared partitions overlap with other partitions.

Like gang-scheduled partitions, space-shared partitions have a priority and an effective priority limit. Each application within a space-shared partition has a priority from 0 to 10, and the partition's priority is the lesser of the effective priority limit and the highest application priority in the partition. Since applications in space-scheduled partitions never overlap, their priorities are never compared with each other. However, the priorities of applications in a space-scheduled partition are important because they determine the partition's priority when compared with other partitions at its own hierarchical level.

Unlike gang-scheduled partitions, space-shared partitions do not have a rollin quantum (since applications never overlap, they never have to be rolled in or out). In effect, the rollin quantum of a space-shared partition is "infinite."

Gang Scheduling

Gang scheduling is a special scheduling technique that coordinates the scheduling of related processes running on separate processors. Gang scheduling is typically used only in the compute partition, or is not used at all (this is determined by your system administrator).

In a partition that uses gang scheduling, the nodes are scheduled so that all the processes in an application are active at the same time. If there are multiple processes per node in the active application, standard scheduling is used to schedule these processes against each other while the application is active.

Partitions that use gang scheduling may contain subpartitions, and may overlap other partitions of any type.

In a partition that uses gang scheduling, not only does each process have a priority, but there is a separate priority for the application as a whole. An application's priority is a number from 0 (low priority) to 10 (high priority). A gang-scheduled partition also has a priority of its own, as well as two other quantities called the *effective priority limit* and the *rollin quantum*:

- A partition's *priority* is the lower of the following:
  - The priority of the highest-priority application or subpartition in the partition.
  - The partition's effective priority limit.
- A partition's *effective priority limit* is a number from 0 to 10 that places an upper limit on the partition's priority. It does not affect the priorities of applications or partitions within the partition.
- A partition's *rollin quantum* is the amount of time each application in the partition is allowed to be active before the system considers running another application instead. The term "rollin quantum" comes from the application being "rolled in" when it is made active, and "rolled out" when it is made inactive.

A gang-scheduled partition's effective priority limit and rollin quantum are set when the partition is created, and do not vary unless you change them with the **chpart** command. A gang-scheduled partition's priority may vary over time, depending on the priorities of the applications and subpartitions in the partition.

A partition that uses standard scheduling does not have an effective priority limit or rollin quantum. It also does not have a numeric priority; instead, its priority is "infinite" (that is, higher than the priority of any gang-scheduled partition or application).

Gang scheduling is performed recursively, partition by partition. For each gang-scheduled partition in the system, starting with the root partition, the operating system examines all the entities (applications and partitions) within the partition:

1. Entities that do not overlap other entities (that is, they have no nodes in common with any other entity in the partition) are simply scheduled to run for the partition's rollin quantum.
2. Where two or more entities overlap, the priorities of the overlapping entities are compared, and the highest-priority entity is scheduled to run for the partition's rollin quantum.
3. If two or more entities overlap and are tied for highest priority, they are scheduled in a round-robin fashion (each takes turns running for one full rollin quantum).
4. If an entity that is scheduled to run is a partition, the operating system examines and schedules the entities in the partition as described above. This process continues recursively as necessary.

At the end of each partition's rollin quantum, the operating system examines and schedules the entities in the partition again.

Note that rules 2 and 3 mean that, when applications or partitions overlap, the one with the highest priority gets one rollin quantum after another until it completes. Entities with lower priorities get no processor time at all until the higher-priority entity has completed. If there is a tie for highest priority, the tied high-priority entities take turns running, but entities with lower priority get no processor time until *all* the high-priority entities complete. Partitions that use standard scheduling always have the highest priority, so if a standard-scheduled partition overlaps a gang-scheduled partition or an application, the standard-scheduled partition always wins.

NOTE

Use of gang scheduling may be limited by the policies of your site.

Your system administrator can require all compute partitions to use space sharing. If gang scheduling is allowed, the administrator can restrict the number of gang-scheduled partitions in the system, can set a minimum rollin quantum, and can restrict the number of applications that can overlap in each gang-scheduled partition. If you try to create a partition that would exceed these restrictions, you see an error message such as "exceeded allocator configuration parameters" or "scheduling parameters conflict with allocator configuration." See your system administrator for information on the policies in force at your site.

Summary of Scheduling Types

Table 2-2 summarizes the differences between the three scheduling types.

Table 2-2. Summary of Scheduling Types

Characteristic	Standard Scheduling	Space Sharing	Gang Scheduling
Scheduling method used within partition	Each process is scheduled by itself using standard UNIX techniques	All processes in an application run at the same time; each application runs until it completes	All processes in an application run at the same time; applications may be rolled in and out
Partitions that typically use this scheduling type	Service partition	All other partitions	Compute partition, or none at all
Restrictions on partition overlap	Partition may not overlap other standard-scheduled partitions	Partition may overlap other partitions (but overlap can lose benefits of space sharing)	Partition may overlap other partitions
Restrictions on subpartition overlap	Subpartitions are not allowed	Subpartitions may not overlap other subpartitions; active subpartitions may not overlap applications	Subpartitions may overlap; maximum depth of overlap can be restricted by system administrator
Restrictions on application overlap	Applications may overlap to any depth	Applications may not overlap other applications or active subpartitions	Applications may overlap; maximum depth of overlap can be restricted by system administrator
Special partition characteristics	Partition priority (always "infinite")	Partition priority, effective priority limit	Partition priority, effective priority limit, rollin quantum

A Scheduling Example

Suppose that a partition has 10 nodes, and an application is currently running on 5 of those nodes. If you attempt to run a new application on 6 nodes of that partition, the results depend on the partition's scheduling type:

- If the partition uses standard scheduling, both applications run at once. Where the applications overlap, the two applications' processes time-share the node. No attempt is made to coordinate when the processes are active with the rest of the application.
- If the partition uses space sharing, the new application fails with the error message "request overlaps with nodes in use" and does not run.
- If the partition uses gang scheduling, the two applications' priorities are compared:
  - If the new application's priority is greater than the old application's, the entire old application is immediately rolled out and the new application starts running. The new application runs until it finishes, then the old application is rolled back in.
  - If the new application's priority is less than the old application's, the entire new application waits until the old application finishes. (During this time it may appear to be "hung.") When the old application finishes, the new application is rolled in and runs until it finishes.
  - If the two applications' priorities are equal, the applications alternate running on each rollin quantum. If one application finishes first, the other runs in every rollin quantum until it finishes.

You can use the **pspart** command to determine which applications are currently running in a partition and what their priorities are, and you can use the command **showpart -f** to determine which nodes in a partition have applications running on them.

Making Partitions

Command Synopsis	Description
mkpart [**-sz** *size* ǀ **-sz** *h***X***w* ǀ **-nd** *nodespec*] [**-nt** *nodetype*] [**-rlx**] [**-ss** ǀ [[**-sps** ǀ **-rq** *time*] [**-epl** *priority*]]] [**-mod** *mode*] *name*	Create a partition.

To create a partition, use the **mkpart** command. You can specify either a relative or an absolute partition pathname for the new partition. The specified new partition must not exist; the parent partition of the new partition must exist and must grant you write permission.

For example, to create a partition called *mypart* whose parent partition is the compute partition, you can use the following command:

```
% mkpart mypart
```

The following command has the same effect, but uses an absolute partition pathname:

```
% mkpart .compute.mypart
```

Specifying the Nodes Allocated to the Partition

The **mkpart** command gives you four ways to specify which nodes are allocated to the new partition:

-sz *size* Creates a partition whose size (number of nodes) is *size*. The **-sz** *size* switch attempts to create a square partition if it can. If this is not possible, it attempts to create a rectangular partition that is either twice as wide as it is high or twice as high as it is wide. If this is not possible, it uses any available nodes. In this case, the nodes allocated to the partition may not be contiguous.

-sz *h***X***w* Creates a contiguous rectangular partition that is *h* nodes high and *w* nodes wide. (You can use an uppercase or lowercase letter **X** between the integers *h* and *w*.)

-nd *nodespec* Creates a partition that consists of exactly the specified nodes, where *nodespec* is one of the following:

x The node whose node number is *x*.

x..*y* The range of nodes from numbers *x* to *y*.

*h***X***w***:***n* The rectangular group of nodes that is *h* nodes high and *w* nodes wide and whose upper left corner is node number *n*. (You can use an uppercase or lowercase letter **X** between the integers *h* and *w*.)

nspec[,*nspec*]... The specified list of nodes, where each *nspec* is a node specifier of the form *x*, *x*..*y*, or *h***X***w***:***n* (no node may appear more than once in this list). Do not put any spaces in this list.

The numbers you use with **-nd** are node numbers within the parent partition, which always range from 0 to one less than the size of the partition.

-nt *nodetype* Creates a partition that consists only of nodes of the specified type. The *nodetype* is the same as the *nodetype* used with the **-nt** switch when running an application, as described under "Running an Application on a Particular Node Type" on page 2-23.

If you don't use the **-sz**, **-nd**, or **-nt** switch, all the available nodes of the parent partition are allocated to the new partition.

You can use at most one **-sz** or **-nd** switch in a single **mkpart** command. You can use **-nt** alone, or with **-sz** or **-nd**. If you use **-nt** without **-sz** or **-nd**, the new partition consists of all the nodes of the specified type in the parent partition. If you use **-nt** together with **-sz** or **-nd**, the new partition consists of the specified nodes of the specified type; if the specified nodes are not all of the specified type, the command fails (see the examples below for more information).

Examples

The following examples all create a partition called *mypart* whose parent partition is the compute partition (that is, the new partition's absolute partition pathname is *.compute.mypart*):

- This command creates a 50-node partition with no specified shape or location:

  ```
  % mkpart -sz 50 mypart
  ```

  The nodes of the new partition are selected from the parent partition by the system, and they may not be contiguous.

- This command creates a partition 10 nodes high and 5 nodes wide:

  ```
  % mkpart -sz 10x5 mypart
  ```

  The position of the new partition within the parent partition is selected by the system, but the new partition is a contiguous rectangle.

- This command creates a partition 10 nodes high and 5 nodes wide located in the upper left corner of the parent partition:

  ```
  % mkpart -nd 10X5:0 mypart
  ```

  The shape and position of the new partition are specified by the user, and the new partition is a contiguous rectangle.

- This command creates a partition that consists of nodes 30 through 79 of the parent partition:

  ```
  % mkpart -nd 30..79 mypart
  ```

  The specific nodes of the partition are specified by the user, and the new partition may or may not be contiguous (its shape depends on the size and shape of the compute partition).

- This command creates a partition that consists of node 0, nodes 3 through 16, and a 5 by 7 node rectangle located at node 21 of the parent partition:

  ```
  % mkpart -nd 0,3..16,5X7:21 mypart
  ```

  The specific nodes of the partition are specified by the user, and the new partition is not contiguous (its shape depends on the size and shape of the compute partition).

- This command creates a partition that consists of all the MP nodes with 32M bytes of memory in the parent partition:

  ```
  % mkpart -nt mp,32mb mypart
  ```

  If there are no 32MB MP nodes in the parent partition, the command fails.

- This command creates a 50-node partition that consists entirely of two-procesor nodes:

  ```
  % mkpart -sz 50 -nt 2proc mypart
  ```

  If there are not at least 50 two-processor nodes in the partition, the command fails.

- This command creates a partition 5 nodes high and 10 nodes wide in the upper left corner of the parent partition, which consists entirely of GP nodes:

  ```
  % mkpart -nd 5x10:0 -nt gp mypart
  ```

  If the specified rectangle does not consist entirely of GP nodes, the command fails.

No matter how you specify the partition's size, nodes are always numbered from 0 to one less than the partition's size. In most cases, they are numbered from left to right and then top to bottom, as discussed under "Nodes Allocated to the Partition" on page 2-35. However, if you use the **-nd** switch, the nodes in the new partition are numbered in the order you specified them in the **-nd** switch. For example, the following command creates a partition that consists of nodes 30 through 79 of the compute partition:

```
% mkpart -nd 79..30 mypart
```

In this case, node 79 of the parent partition is node 0 of the new partition; node 78 of the parent partition is node 1 of the new partition; and so on to node 30 of the parent partition, which is node 49 of the new partition.

Relaxing Partition Size

No matter how you specify the partition's size, if any of the nodes you request is not available, the **mkpart** command fails with an error message and no partition is created. Whether or not a node is available is determined by the parent partition's scheduling type and whether or not the node is already in use; for example, if the partition does not permit overlapping subpartitions, any node that is already allocated to a subpartition is not available. See "Scheduling Characteristics" on page 2-39 for more information. A node can also be unavailable due to a software or hardware problem; see "Unusable Nodes" on page 2-37.

You can use the switch **-rlx** to relax the requirement that the exact specified number of nodes must be available. When you use **-rlx**, the new partition may consist of *fewer* nodes than you requested. In other words, the new partition consists of *as many nodes as possible*, *up to* the requested number of nodes. However, there must be at least one node available or the **mkpart** command still fails.

NOTE

-rlx can be used to relax the default size, the **-sz** *size* switch, or the **-nd** switch. It cannot be used together with the switch **-sz** *h***X***w*.

For example, the following command creates a partition called *mypart* that consists of every available node in the compute partition:

```
% mkpart -rlx mypart
```

The following command creates a partition that consists of *up to* five nodes of the compute partition. If less than five nodes are available, the application consists of all the available nodes:

```
% mkpart -sz 5 -rlx mypart
```

In either of the above cases, if *no* nodes are available in the compute partition, the command fails.

The following command creates a partition that consists of *up to* a 3-by-3-node rectangle of nodes located in the upper left corner of the compute partition. If any of those nodes is not available, the partition consists of the remaining nodes of that rectangle.

```
% mkpart -nd 3x3:0 -rlx mypart
```

In this case, if no nodes are available *in the specified rectangle*, the command fails.

If you use **-rlx** together with **-nt** it relaxes the *number* of nodes requested, but does not relax the *type* of nodes requested. The new partition will always consist entirely of nodes of the type specified by **-nt**, but it may consist of *fewer* nodes than specified by **-sz**, **-nd**, or the default size. The command fails unless there is at least one node of the specified type available.

For example, the following command creates a partition that is *up to* 5 nodes high and 10 nodes wide, is located in the upper left corner of the compute partition, and consists entirely of GP nodes. If any of those nodes is not available or is not a GP node, the partition consists of the available GP nodes in that rectangle.

```
% mkpart -nd 5x10:0 -nt gp -rlx mypart
```

If no GP nodes are available in the specified rectangle, the command fails.

Specifying Protection Modes

The **mkpart** command gives you two ways to specify the protection modes of the new partition:

-mod *nnn* Creates a partition whose protection modes are specified by the three-digit octal number *nnn*, as used by the **chmod** command (see **chmod** in the *OSF/1 Command Reference* for more information).

-mod *string* Creates a partition whose protection modes are specified by the nine-character string *string*. The string must have the form `rwxrwxrwx`, where a letter (`r`, `w`, or `x`) represents a permission granted and a dash (`-`) represents a permission denied, as displayed by the command **ls -l** (see **ls** in the *OSF/1 Command Reference* for more information).

You can use at most one **-mod** switch in a single **mkpart** command. If you don't use the **-mod** switch, the new partition is given the same protection modes as its parent partition.

For example, the following command creates a partition that is readable, writable, and executable by you; readable and executable by your group, and only readable by others:

```
% mkpart -mod rwxr-xr-- mypart
```

The following command has the same effect, but uses an octal number:

```
% mkpart -mod 754 mypart
```

Specifying Scheduling Characteristics

The **mkpart** command gives you three switches to specify the scheduling characteristics of the new partition:

-ss Creates a partition that uses standard scheduling.

-ss cannot be used with **-sps**, **-rq** or **-epl**.

-rq *time* Creates a partition that uses gang scheduling with a rollin quantum of *time*, where *time* is one of the following:

n *n* milliseconds (if *n* is not a multiple of 100, it is silently rounded up to the next multiple of 100).

*n***s** *n* seconds.

*n***m** *n* minutes.

*n***h** *n* hours.

0 "Infinite" time: once rolled in, an application runs until it exits.

The maximum rollin quantum is 24 hours; the minimum rollin quantum for your system is determined by your system administrator.

-rq cannot be used with **-ss** or **-sps**. **-rq** can be used with or without **-epl**; if you use **-rq** without **-epl**, the new partition is a gang-scheduled partition with the same effective priority limit as its parent partition.

If gang-scheduled partitions are not allowed at your site, or creating a gang-scheduled partition would exceed the maximum number of gang-scheduled partitions, any attempt to create a partition with **-rq** fails.

-sps Creates a partition that uses space sharing.

-sps cannot be used with **-ss** or **-rq**. **-sps** can be used with or without **-epl**; if you use **-sps** without **-epl**, the new partition is a space-shared partition with the same effective priority limit as its parent partition.

-epl *priority* Creates a partition with an effective priority limit of *priority*, where *priority* is an integer from 0 to 10 inclusive (0 is low priority, 10 is high priority).

-epl cannot be used together with -ss. If you use **-epl** without either **-sps** or **-rq**, the results depend on the scheduling type of the parent partition:

- If the parent partition is a space-shared partition, the new partition is a space-shared partition with the specified effective priority limit.

- If the parent partition is a gang-scheduled partition, the new partition is a gang-scheduled partition with the specified effective priority limit and the same rollin quantum as its parent. If this would exceed the maximum number of gang-scheduled partitions, the new partition is a space-shared partition instead.

If you don't use the **-ss**, **-rq**, or **-sps** switch, the new partition uses the same scheduling technique, rollin quantum, and effective priority limit as its parent partition.

For example, the following command creates a partition that uses standard scheduling:

```
% mkpart -ss mypart
```

The following command creates a partition that uses gang scheduling with a rollin quantum of 10 seconds and the same effective priority limit as its parent partition:

```
% mkpart -rq 10s mypart
```

The following command creates a partition that uses space sharing with the same effective priority limit as its parent partition:

```
% mkpart -sps mypart
```

The following command creates a partition that uses gang scheduling with a rollin quantum of 5 minutes and an effective priority limit of 6:

```
% mkpart -rq 5m -epl 6 mypart
```

Removing Partitions

Command Synopsis	Description
rmpart [**-f**] [**-r**] *partition*	Remove a partition.

To remove an existing partition, use the **rmpart** command. You must have write permission on the parent partition of the partition to be removed. You can specify the partition to be removed with either a relative or an absolute partition pathname.

For example, to remove the partition called *mypart*, whose parent partition is the compute partition, you can use the following command:

```
% rmpart mypart
```

The following command has the same effect, but uses an absolute partition pathname:

```
% rmpart .compute.mypart
```

Removing Partitions Containing Running Applications

If you specify a partition that contains any running applications, you see an error message and the partition is not removed. You can force **rmpart** to remove a partition that contains running applications with the **-f** switch. When you use the **-f** switch, **rmpart** terminates all the applications running in the specified partition and then removes it.

For example, if there are applications running in *mypart*, use the following command to terminate the applications and remove the partition:

```
% rmpart -f mypart
```

Removing Partitions Containing Subpartitions

If you specify a partition that contains any subpartitions, you see an error message and the partition is not removed. You can force **rmpart** to remove a partition that contains subpartitions with the **-r** switch. When you use the **-r** switch, **rmpart** recursively removes all the subpartitions in the specified partition (and their sub-subpartitions, and so on) and then removes the specified partition.

For example, if there are subpartitions in *mypart*, use the following command to remove *mypart* and all its subpartitions:

```
% rmpart -r mypart
```

rmpart -r is an "all or nothing" operation. If any subpartitions cannot be removed, the command fails and no subpartitions are removed.

The **-r** switch does not imply **-f**. If *mypart* or any of its subpartitions contains any running applications, you see an error message and none of the partitions are removed. You can force **rmpart** to remove a partition that contains subpartitions and running applications by using the **-r** and **-f** switches together. When you use both these switches, **rmpart** terminates all the applications running in the specified partition and its subpartitions, removes all the subpartitions in the specified partition, and then removes the specified partition.

Showing Partition Characteristics

Command Synopsis	Description
showpart [**-f**] [**-l** | **-p**] [**-w**] [**-nt** *nodetype*] [*partition*]	Show the characteristics of a partition.

To show the characteristics of a partition, use the **showpart** command. You can specify the partition with either a relative or an absolute partition pathname. If you don't specify a partition, **showpart** shows the characteristics of your default partition (see "Using the Default Partition" on page 2-13). In either case, you must have read permission on the specified partition.

For example, to show the characteristics of the partition called *mypart*, whose parent partition is the compute partition, you can use the following command:

```
% showpart mypart
 USER      GROUP    ACCESS  SIZE      FREE  RQ   EPL
 smith     eng         777     9         5  15m    5
     +---------+
    0| . . . . |
    4| . * * * |
    8| . * * * |
   12| . * * * |
     +---------+
```

The following command has the same effect, but uses an absolute partition pathname:

```
% showpart .compute.mypart
```

The columns at the top of the **showpart** output have the following meanings:

USER The owner of the partition, in this case *smith.*

GROUP The group of the partition, in this case *eng.*

ACCESS The access permissions, expressed as an octal number, in this case 777 (which represents the permissions `rwxrwxrwx`).

SIZE The number of nodes in the partition, in this case 9.

FREE The number of free nodes in the partition, in this case 5 (see "Showing Free Nodes" on page 2-55 for more information on free nodes).

RQ The rollin quantum or scheduling type of the partition, as follows:

- The partition uses standard scheduling.

SPS The partition uses space sharing.

time The partition uses gang scheduling with a rollin quantum of *time*. The *time* is expressed as a number followed by an optional letter: no letter for milliseconds, **s** for seconds, **m** for minutes, or **h** for hours.

In this case, the partition is a gang-scheduled partition with a rollin quantum of 15 minutes.

EPL The effective priority limit of the partition, in this case 5, or a dash (-) for a standard-scheduled partition.

See "Partition Characteristics" on page 2-34 for information on these partition characteristics.

The rectangular picture at the bottom of the **showpart** output shows the size, shape, and position of the specified partition within the system:

- The large rectangle represents the root partition. In this case, the root partition is 4 nodes high and 4 nodes wide.

- The numbers to the left of the rectangle show the node numbers of the nodes in the first column of each row. In this case, the first node in the top row is node 0, the first node in the second row is node 4, the first node in the third row is node 8, and the first node in the bottom row is node 12.

- Asterisks (*) within the rectangle represent nodes that are allocated to the specified partition; periods (.) represent other nodes. In this case, *mypart* consists of nodes 5–7, 9–11, and 13–15 of the root partition.

- If you see a dash (-) or an X within the rectangle, it represents an unusable node that is allocated to the specified partition. You cannot run any applications or allocate any partitions using this node. See "Unusable Nodes" on page 2-37 for more information.

Showing Free Nodes

The output of **lspart** or **showpart** includes the number of free nodes in the FREE column. A node is *free* if no application is running on that node and no subpartition in which any applications are running includes that node. (Note that *all* the nodes of a subpartition are considered busy if an application is running *anywhere* in the subpartition, or in any of its sub-subpartitions. This occurs because partitions are scheduled recursively.)

You can use the **-f** switch of **showpart** to see *which* nodes are free. The output of **showpart -f** is the same as the regular **showpart** output, except that free nodes are shown as an F instead of an asterisk.

For example, the following command shows the free nodes in the partition called *mypart*:

```
% showpart -f mypart
 USER      GROUP     ACCESS  SIZE        FREE  RQ   EPL
 smith     eng          777     9           5  15m    5
    +---------+
   0| . . . . |
   4| . * * * |
   8| . * F F |
  12| . F F F |
    +---------+
```

In this case, *mypart* has five free nodes: nodes 4, 5, 6, 7, and 8 of the partition.

Showing Node Attributes

On some Paragon systems, not all the nodes in the compute partition have the same hardware. For example, some nodes may have more memory than others, or some nodes may have I/O interfaces that the others do not. The hardware characteristics of each node are described by a comma-separated series of strings called *attributes*. See "Running an Application on a Particular Node Type" on page 2-23 for information on node attributes.

You can use the **-l** switch of **showpart** to list the attributes of the nodes in the partition. The output of **showpart -l** shows the attributes of every node in the partition; it also includes an ATTR column that lists the attributes that all the nodes in the partition have in common.

For example, the following command shows the node attributes of the partition called *mypart*:

```
% showpart -l mypart
 USER      GROUP     ACCESS  SIZE        FREE  RQ   EPL  ATTR
 smith     eng          777     9           5  15m    5  2proc,MP

0..2,4,5 2proc,64mb,MP
3,6..8 2proc,128mb,MP

    +---------+
   0| . . . . |
   4| . * * * |
   8| . * * * |
  12| . * * * |
    +---------+
```

In this case, *mypart* has five two-processor MP nodes with 64M bytes of memory (nodes 0, 1, 2, 4, and 5) and four two-processor MP nodes with 128M bytes of memory (nodes 3, 6, 7, and 8). The attributes that all nodes have in common are that they are all two-processor MP nodes; these attributes are shown in the ATTR column.

Showing Node Attributes with Root Node Numbers

The node numbers shown in the middle section of the **-l** output are node numbers relative to the specified partition (*logical* node numbers). You can also use the **-p** switch to see the same information with node numbers relative to the root partition (*physical* node numbers). These node numbers correspond to the numbers to the left of the rectangle and reflect the node's physical position within the system. The output of **showpart -p** is otherwise identical to **showpart -l**. The **-l** and **-p** switches are mutually exclusive.

For example, the following command shows the node attributes of the partition called *mypart* with node numbers relative to the root partition:

```
% showpart -p mypart
 USER     GROUP    ACCESS  SIZE       FREE  RQ   EPL  ATTR
 smith    eng         777     9          5  15m    5  2proc,MP

5..7,9,10 2proc,64mb,MP
11,13..15 2proc,128mb,MP

    +---------+
   0| . . . . |
   4| . * * * |
   8| . * * * |
  12| . * * * |
    +---------+
```

This display is the same as the example on page 2-56, except the node numbers are displayed as root-partition node numbers. The five two-processor MP nodes with 64M bytes of memory are nodes 5, 6, 7, 9, and 10, and the four two-processor MP nodes with 128M bytes of memory are nodes 11, 13, 14, and 15.

Showing Nodes Having Certain Attributes

The **-l** or **-p** switch of **showpart** lists the attributes of each node, but it is also useful to see the positions of nodes having certain attributes within the partition. To do this, use the switch **-nt** *nodetype*, where *nodetype* is a string describing the desired nodes' attributes, as described under "Specifying Node Attributes" on page 2-25. Nodes in the partition having the attributes specified in the *nodetype* string are shown with an asterisk (`*`); other nodes within the partition are shown with a lowercase letter O (`o`).

For example, the following command shows the positions of the 64M -byte nodes in the partition called *mypart*:

```
% showpart -nt 64mb mypart
 USER      GROUP     ACCESS  SIZE        FREE  RQ   EPL
 smith     eng          777     9           5  15m    5
    +---------+
   0|  . . . .  |
   4|  . * * *  |
   8|  . o * *  |
  12|  . o o o  |
    +---------+
```

As in the previous two examples, *mypart* has five nodes with 64M bytes of memory (nodes 0, 1, 2, 4, and 5). These nodes are shown with asterisks. The other four nodes do not have 64M bytes of memory; these nodes are shown with o characters. (Note that you do not know anything about the nodes shown with o characters except that they do not have 32M bytes of memory. To find out more about the attributes of these nodes, you would have to use the **-l** or **-p** switch. **-l** or **-p** can be used together with **-nt** if desired.)

If you use **-nt** together with **-f**, free nodes that match the *nodetype* string are shown with a capital F, while free nodes that do not match are shown with a lowercase f. For example:

```
% showpart -f -nt 32mb mypart
 USER      GROUP     ACCESS  SIZE        FREE  RQ   EPL
 smith     eng          777     9           5  15m    5
    +---------+
   0|  . . . .  |
   4|  . * * *  |
   8|  . o F F  |
  12|  . f f f  |
    +---------+
```

In this case:

- Nodes 0, 1, and 2 of the partition (shown as asterisks) are 32MB nodes that are not free.
- Node 3 of the partition (shown as a lowercase letter o) is not a 32MB node and is not free.
- Nodes 4 and 5 of the partition (shown as capital Fs) are 32MB nodes and are free.
- Nodes 6, 7, and 8 of the partition (shown as lowercase fs) are not 32MB nodes and are not free.

Showing Partitions with Cabinet Information

You can use the **-w** switch of **showpart** to see which nodes are in which cabinet and to easily determine a node's number relative to the root partition. This is particularly useful on larger systems.

The output of **showpart -w** is the similar to the regular **showpart** output, with the addition that the command shows the cabinet and backplane location of the partition's nodes. The picture is divided into columns and rows. The columns indicate the system's cabinets and the rows indicate the system's backplanes. The top of the rectangular picture shows the offsets from the numbers in the left column. The plus sign (+) indicates that the number is an offset. The offsets are in multiples of four. The bottom of the picture shows the cabinet numbers. For example, Cab 2 indicates that the column is for the nodes in cabinet 2. The following example shows the locations of the nodes in the compute partition in a three-cabinet system.

```
% showpart -w
USER     GROUP    ACCESS  SIZE         FREE  RQ   EPL
root     daemon      777   128          128   0     5

    |+0       |+4       |+8       |
    +-----------------------------+
   0| . * * * | * * * * | * . . . |
  12| . * * * | * * * * | * . . . |
  24| . * * * | * * * * | * . . . |
  36| . * * * | * * * * | * . . . |
    +-----------------------------+
  48| . * * * | * * * * | * . . . |
  60| . * * * | * * * * | * . . . |
  72| . * * * | * * * * | * . . . |
  84| . * * * | * * * * | * . . . |
    +-----------------------------+
  96| . * * * | * * * * | * . . . |
 108| . * * * | * * * * | * . . . |
 120| . * * * | * * * * | * . . . |
 132| . * * * | * * * * | * . . . |
    +-----------------------------+
 144| . * * * | * * * * | * . . . |
 156| . * * * | * * * * | * . . . |
 168| . * * * | * * * * | * . . . |
 180| . * * * | * * * * | * . . . |
    +-----------------------------+
    | Cab   2 | Cab   1 | Cab   0 |
```

The rectangular picture at the bottom of the **showpart** output shows there are three cabinets in the system. The top line of the picture shows the cabinet offsets in multiples of four; the bottom line shows the cabinet numbers. Using the offsets, the top left-hand node in the compute partition is node 2 and the bottom right-hand node in the compute partition is node number 188 (180 + 8).

Summary of Symbols

The symbols used in the output of the **showpart** command are summarized in Table 2-3.

Table 2-3. Symbols Used in showpart Output

Symbol	Meaning
.	Node not belonging to partition. (All other symbols represent nodes belonging to the partition.)
*	Without **-f** or **-nt**: Any node belonging to partition. With **-f**: Node that is not free. With **-nt**: Node matching specified attributes. With **-f** and **-nt**: Free node matching specified attributes.
F	With **-f**: Free node. With **-f** and **-nt**: Free node matching specified attributes.
o	With **-nt**: Node not having specified attributes.
f	With **-f** and **-nt**: Free node not matching specified attributes.
-	Empty slot (unusable node).
X	Node that failed to boot (unusable node).

Listing Subpartitions

Command Synopsis	Description
lspart [**-r**] [**-l**] [**-p**] [*partition*]	List the subpartitions of a partition.

To list the subpartitions of a partition with their characteristics, use the **lspart** command. You can specify the partition with either a relative or an absolute partition pathname. If you don't specify a partition, **lspart** lists the subpartitions of your default partition (see "Using the Default Partition" on page 2-13). In either case, you must have read permission on the specified partition.

For example, to list the subpartitions of the partition called *mypart*, whose parent partition is the compute partition, you can use the following command:

```
% lspart mypart
 USER      GROUP     ACCESS  SIZE       FREE  RQ    EPL  PARTITION
 chris     eng          777    16          4  15m     3  mandelbrot
 chris     eng          777    16         16   -      -  debug
 pat       mrkt         755     4          0  SPS    10  slalom
 *         *              *     *          *   *      *  private
```

The following command has the same effect, but uses an absolute partition pathname:

```
% lspart .compute.mypart
```

The columns in the output of **lspart** are the same as the top part of the output of **showpart** (see "Showing Partition Characteristics" on page 2-54), with the addition of the partition name. In this case, *mypart* has four subpartitions: *mandelbrot*, *debug*, *slalom*, and *private*.

- *mandelbrot* is owned by user *chris* in group *eng*; it has permissions `rwxrwxrwx` and a size of 16 nodes, of which 4 are free (see "Showing Free Nodes" on page 2-55 for more information on free nodes). It is a gang-scheduled partition with a rollin quantum of 15 minutes and an effective priority limit of 3.

- *debug* is also owned by user *chris* in group *eng*; it has permissions `rwxrwxrwx` and a size of 16 nodes, of which all 16 are free. It is a standard-scheduled partition, so it has no rollin quantum or effective priority limit.

- *slalom* is owned by user *pat* in group *mrkt*; it has permissions `rwxr-xr-x` and a size of 4 nodes, of which none are free. It is a space-shared partition with an effective priority limit of 10.

- *private*'s access permissions do not grant you read permission, so all its characteristics are shown as asterisks (`*`).

If you see two numbers separated by a slash in the SIZE column, it indicates that one or more of the nodes allocated to the indicated partition is unusable. For example:

```
% lspart mypart
 USER      GROUP     ACCESS  SIZE       FREE  RQ    EPL  PARTITION
 chris     eng          777    14 / 16    10  15m     3  mandelbrot
```

This indicates that there are 16 nodes allocated to *mandelbrot*, but 2 of them are currently unusable. You cannot run any applications or allocate any partitions using unusable nodes. See "Unusable Nodes" on page 2-37 for more information.

Recursively Listing Subpartitions

To recursively list all of a partition's subpartitions, sub-subpartitions, and so on, use the **-r** switch. For example:

```
% lspart -r mypart
   USER     GROUP     ACCESS  SIZE    FREE  RQ    EPL  PARTITION
.compute.mypart:
   chris    eng          777    16       4  15m     3  mandelbrot
   chris    eng          777    16      16   -      -  debug
   pat      mrkt         755     4       0  SPS    10  slalom
   *        *              *     *       *   *      *  private
.compute.mypart.mandelbrot:
   chris    eng          777    16      16  15m    10  hi_pri
   chris    eng          777    16      16  15m     1  lo_pri
```

The **lspart -r** output reveals that *mypart.mandelbrot* has two subpartitions, *hi_pri* and *lo_pri*, neither of which has any sub-subpartitions, and that *slalom* and *debug* have no subpartitions. No information is available on the subpartitions of *private* (if any), because *private* does not grant you read permission.

NOTE

If you specify a partition that has no subpartitions, **lspart** produces no output.

For example, since *mypart.slalom* has no subpartitions, an **lspart** command on this partition gives no output:

```
% lspart mypart.slalom
%
```

To get information about *mypart.slalom* itself, use the **showpart** command.

Listing Node Attributes of Subpartitions

You can use the **-l** switch of **lspart** to list the attributes of the nodes in each subpartition. The output of **lspart -l** shows the attributes of every node in the partition; it also includes an ATTR column that lists the attributes that all the nodes in the subpartition have in common. For example:

```
% lspart -l mypart
 USER     GROUP    ACCESS  SIZE      FREE  RQ   EPL  PARTITION   ATTR
 chris    eng         777    16         4  15m    3  mandelbrot  1proc

0..15 1proc,16mb,GP

 chris    eng         777    16        16   -     -  debug       16mb

0..3,6,12 1proc,16mb,MP
4,5,7..11,13..15 1proc,16mb,GP

 pat      mrkt        755     4         0 SPS    10  slalom      1proc,MP

0 1proc,32mb,MP
1..3 1proc,64mb,MP

 *        *             *     *         *   *     *  private
```

In this example, **lspart -l** displays the node numbers relative to subpartition

- *mandelbrot* consists of 16 one-processor GP nodes with 16M bytes of memory. Because all nodes in *mandelbrot* have the same attributes, this is shown both in the ATTR column and in the list of node attributes following the partition.

- *debug* consists of 6 two-processor MP nodes with 16M bytes of memory (nodes 0, 1, 2, 3, 6, and 12 of *debug*) and 10 one-processor GP nodes with 16M bytes of memory (the remaining nodes). The only thing all these nodes have in common is that they all have 16M bytes of memory; this is shown in the ATTR column.

- *slalom* consists of four two-processor MP nodes. Node 0 of *slalom* has 32M bytes of memory, and nodes 1, 2, and 3 have 64M bytes of memory. The ATTR column shows the common attributes, which is that they are two-processor MP nodes.

- *private*'s access permissions do not grant you read permission, so all its characteristics are shown as asterisks (`*`) and no node attributes are shown.

Listing Node Attributes with Root Node Numbers

The node numbers shown with the node attributes for each subpartition of the **-l** output are node numbers relative to the specified subpartition (*logical* node numbers). You can also use the **-p** switch to see the same information with node numbers relative to the root partition (*physical* node numbers). These node numbers reflect the node's physical position within the system. The output of **lspart -p** is otherwise identical to **lspart -l**. The **-l** and **-p** switches are mutually exclusive.

```
% lspart -p mypart
 USER     GROUP     ACCESS  SIZE      FREE  RQ    EPL  PARTITION    ATTR
 chris    eng          777    16         4  15m     3  mandelbrot   1proc

49..52,57..60,65..68,73..76 1proc,16mb,GP

 chris    eng          777    16        16   -      -  debug        16mb

42..46,53,54,61,62,69,70,77,78,81..83 1proc,16mb,GP

 pat      mrkt         755     4         0 SPS     10  slalom       1proc,MP

84..86,89 1proc,64mb,MP


*        *          *       *          *    *     *  private
```

In this example, **lspart -l** displays the same information as the example on page 2-63, except the node numbers are relative to the root partition.

Listing the Applications in a Partition

Command Synopsis	Description
pspart [**-r**] [*partition*]	List the applications in a partition.

To list the applications in a partition, with information about the rollin/rollout status of each, use the **pspart** command. You can specify the partition with either a relative or an absolute partition pathname. If you don't specify a partition, **pspart** lists the applications in your default partition (see "Using the Default Partition" on page 2-13). In either case, you must have read permission on the specified partition.

For example, to list the applications in the partition *mypart*, whose parent partition is the compute partition, you can use the following command:

```
% pspart mypart
    PGID  USER       SIZE PRI START       TIME ACTIVE   TOTAL TIME COMMAND
   12345  pat         256   5 11:42:20      45.00  75%     0:04:41 mag -sz 256
   23456  chris        67   4   Jan 21          -    -     0:12.30 boggle
   34567  smith       192  10 02:21:51    0:01:00 100%     2:12:03 myfft
```

The following command has the same effect, but uses an absolute partition pathname:

```
% pspart .compute.mypart
```

The columns in the output of **pspart** have the following meanings:

`PGID` The process group ID of the application (see "Process Groups" on page 4-27 for more information).

The process group ID of an application is always the same as the process ID of the application's controlling process. This means that you can use this number with the **kill** command to kill the application; for example, given the **pspart** output above, the command **kill 34567** would kill the application *myfft*.

`USER` The login name of the user who invoked the application.

`SIZE` The number of nodes allocated to the application from the partition (see "Specifying Application Size" on page 2-15 for more information).

`PRI` The application's priority (see "Specifying Application Priority" on page 2-18 for more information).

`START` The time the application was started. If the application was started more than 24 hours ago, the date it was started is shown instead.

`TIME ACTIVE` The amount of time the application has been active (rolled in) in the current rollin quantum (see "Gang Scheduling" on page 2-42 for more information). The time active is shown both as an absolute time (in the format *minutes* : *seconds* . *milliseconds* for times less than one minute or *hours* : *minutes* : *seconds* for times of one minute or more) and as a percentage of the partition's rollin quantum. If the application is not active in the current rollin quantum, a dash (`-`) is shown for both quantities. If the partition uses space sharing, the time shown is the total amount of time the application has been running and the percentage is always 100%.

In the example above, the partition *mypart* is a gang-scheduled partition with a rollin quantum of one minute. The application *mag* has been active for 45 seconds, or 75% of the rollin quantum; the application *boggle* is not currently active; and the application *myfft* has been active for one minute, or 100% of the rollin quantum.

TOTAL TIME The total amount of time the application has been rolled in since it was started, in the format *minutes* : *seconds* . *milliseconds* or *hours* : *minutes* : *seconds*. If the partition uses space sharing, the TOTAL TIME is always the same as the TIME ACTIVE.

In the example above, the application *mag* has been active for a total of 4 minutes and 41 seconds; the application *boggle* has been active for a total of 12.30 seconds; and the application *myfft* has been active for a total of 2 hours, 12 minutes, and 3 seconds.

COMMAND The command line by which the application was invoked.

Applications in Subpartitions

If there are any applications running in subpartitions of the specified partition, the subpartitions appear in the output of **pspart** as follows:

```
% pspart mypart
     PGID  USER      SIZE PRI START      TIME ACTIVE   TOTAL TIME COMMAND
    12345  pat        256   5 11:42:20     45.00  75%    0:04:41 mag -sz 256
    23456  chris       67   4   Jan 21          -    -    0:12.30 boggle
    34567  smith      192  10 02:21:51    0:01:00 100%    2:12:03 myfft
Active Partitions
    OWNER  GROUP     SIZE PRI START      TIME ACTIVE   TOTAL TIME NAME
smith      eng         64   6 09:16:30          -    -    1:18.10 subpart
```

The columns for the list of active partitions have the following meanings:

OWNER The owner of the subpartition.

GROUP The group of the subpartition.

SIZE The size of the subpartition (note that all nodes of a subpartition containing an active application are considered active, even if not all the nodes in the subpartition are actually in use by applications).

PRI The current priority of the subpartition (this is the highest priority of all the applications in the subpartition or the subpartition's effective priority limit, whichever is lower).

START The time or date when the oldest application in the subpartition was started.

TIME ACTIVE The amount of time the subpartition has been active (rolled in) in the current rollin quantum.

TOTAL TIME The total amount of time the subpartition has been rolled in since it was started.

NAME The name of the subpartition.

See "Scheduling Characteristics" on page 2-39 for more information on how subpartitions are scheduled.

Recursively Listing Applications in Subpartitions

If there are applications running in a subpartition, the output of **pspart** normally shows only that the subpartition is active. To list the applications in subpartitions (and, recursively, in sub-subpartitions and so on), use the **-r** switch. For example:

```
% pspart -r mypart
mypart:
     PGID  USER      SIZE PRI START      TIME ACTIVE   TOTAL TIME COMMAND
    12345  pat        256   5 11:42:20     45.00  75%    0:04:41 mag -sz 256
    23456  chris       67   4   Jan 21         -    -    0:12.30 boggle
    34567  smith      192  10 02:21:51    0:01:00 100%    2:12:03 myfft
Active Partitions
    OWNER  GROUP     SIZE PRI START      TIME ACTIVE   TOTAL TIME NAME
  smith      eng         64   6 09:16:30         -    -    1:18.10 subpart
mypart.subpart:
     PGID  USER      SIZE PRI START      TIME ACTIVE   TOTAL TIME COMMAND
    45678  smith       56   7 09:16:30         -    -    1:18.10 span
```

In this case, the **-r** switch shows that the subpartition *subpart* has one application, *span*, which is running on 56 nodes of the subpartition. (Even though the application is not running on every node of the subpartition, whenever the application is rolled in the entire subpartition is rolled in. This occurs because subpartitions are scheduled recursively, as discussed under "Gang Scheduling" on page 2-42.)

Listing Applications With Core Dumps

If there are applications in a subpartition that have one or more processes fault and dump core, the string "(core dump)" is appended to the application name.

```
% pspart mypart
 PGID USER SIZE PRI START    TIME ACTIVE   TOTAL TIME COMMAND
  42   pat   12   5  12:51:22 0:06.30 63%   0:02:42    myapp (core dump)
```

In this example, the application *myapp* is running on some (not necessarily all) nodes in the partition *mypart.* The display indicates that the application *myapp* has one or more processes dumping core.

Changing Partition Characteristics

Command Synopsis	Description
chpart [**-rq** *time* | **-sps**] [**-epl** *priority*] [**-nm** *name*] [**-mod** *mode*] [**-g** *group*] [**-o** *owner*[**.** *group*]] *partition*	Change certain partition characteristics.

To change the characteristics of a partition, use the **chpart** command. The permissions required depend on the switches you use. You can specify the partition with either a relative or an absolute partition pathname.

chpart can change the following partition characteristics:

- Rollin quantum.
- Effective priority limit.
- Partition name.
- Protection modes.
- Owner and group.
- Scheduling type (space-shared to gang-scheduled, or gang-scheduled to space-shared with certain limitations; a partition cannot be changed to or from standard scheduling).

A partition's size and parent partition are determined when the partition is created and cannot be changed.

The switches of **chpart**, which can be used together or separately and in any order (except as noted below), are similar to the corresponding switches of **mkpart**:

-rq *time* Changes the partition to a gang-scheduled partition with a rollin quantum of *time*, where *time* is one of the following:

n	*n* milliseconds (if *n* is not a multiple of 100, it is rounded up to the next multiple of 100).
*n***s**	*n* seconds.
*n***m**	*n* minutes.
*n***h**	*n* hours.

0 "Infinite" time: once rolled in, an application runs until it exits.

The maximum rollin quantum is 24 hours; the minimum rollin quantum for your system is determined by your system administrator.

-rq can be used only on a gang-scheduled or space-shared partition, and cannot be used together with **-sps**. To use **-rq**, you must have write permission on the specified partition.

-sps Changes the partition to a space-shared partition.

-sps can be used only on a space-shared or gang-scheduled partition, and cannot be used together with **-rq**. If the partition is currently gang-scheduled, it must not contain any overlapping subpartitions or any applications. To use **-sps**, you must have write permission on the specified partition.

-epl *priority* Changes the partition's effective priority limit to *priority*, where *priority* is an integer from 0 to 10 inclusive.

-epl can be used only on a gang-scheduled or space-shared partition. To use **-epl**, you must have write permission on the specified partition.

-nm *name* Changes the partition's name to *name*, where *name* is a valid partition name (a string of any length containing only uppercase letters, lowercase letters, digits, and underscores). To use **-nm**, you must have write permission on the parent partition of the specified partition.

Note that **-nm** can only change the partition's name "in place;" there is no way to move a partition to a different parent partition.

-mod *nnn* Changes the partition's protection modes to the value specified by the three-digit octal number *nnn*. To use **-mod**, you must be the owner of the specified partition or the system administrator.

-mod *string* Changes the partition's protection modes to the value specified by the nine-character string *string*. The string must have the form `rwxrwxrwx`, where a letter (`r`, `w`, or `x`) represents a permission granted and a dash (`-`) represents a permission denied. To use **-mod**, you must be the owner of the specified partition or the system administrator.

-g *group* Changes the partition's group to *group*. The *group* can be either a group name or a numeric group ID. To use **-g**, you must be the owner of the specified partition *and* a member of the specified new group, or you must be the system administrator.

-o *owner*[**.***group*] Changes the partition's owner to *owner*. If **.***group* is specified, also changes the partition's group to *group*. The *owner* and *group* can be either user/group names or numeric user/group IDs. To use **-o**, you must be the system administrator.

For example, the following command changes the rollin quantum of *mypart* to 20 minutes:

```
% chpart -rq 20m mypart
```

The following command changes *mypart* to a space-shared partition:

```
% chpart -sps mypart
```

The following command changes the effective priority of *mypart* to 2:

```
% chpart -epl 2 mypart
```

The following command changes the protection modes of *mypart* so that it is readable, writable, and executable by the owner but not by anyone else:

```
% chpart -mod rwx------ mypart
```

The following command has the same effect as the previous three commands combined, but uses an absolute partition pathname and an octal protection mode specifier:

```
% chpart -epl 2 -rq 20m -mod 700 .compute.mypart
```

The following command changes the owner of *mypart* to *smith*, but does not affect its group:

```
% chpart -o smith mypart
```

The following command changes the group of *mypart* to *support*, but does not affect its ownership:

```
% chpart -g support mypart
```

The following command changes the owner of *mypart* to *smith* and the group to *support*:

```
% chpart -o smith.support mypart
```

The following command changes the name of *mypart* to *newpart*:

```
% chpart -nm newpart mypart
```

The following command also changes the name of *mypart* to *newpart*, but uses an absolute partition pathname:

```
% chpart -nm newpart .compute.mypart
```

Note that the new name is specified as a name only, not a pathname.

Using Message-Passing System Calls 3

Introduction

Message passing is the standard means of communication among processes in operating system. As independent processor/memory pairs, the nodes do not share physical memory. If the node processes need to share information, they can do so by passing messages. The calls described in this chapter let your programs send and receive messages.

This chapter introduces the message-passing system calls and includes the following sections:

- Process characteristics.
- Message characteristics.
- Names of send and receive calls.
- Synchronous send and receive.
- Asynchronous send and receive.
- Probing for pending messages.
- Getting information about pending or received messages.
- Message passing with Fortran commons.
- Treating a message as an interrupt.
- Extended receive and probe.
- Global operations.

Within each section, the calls are discussed in order of increasing complexity. That is, the "base" calls are discussed first, and the "extended" calls are discussed later.

Each section includes numerous examples in both C and Fortran. A call description at the beginning of each section or subsection gives a language-independent synopsis (call name, parameter names, and brief description) of each call discussed in that section. Differences between C and Fortran are noted where applicable. See Appendix A for information on call and parameter types; see the *Paragon™ System C Calls Reference Manual* or the *Paragon™ System Fortran Calls Reference Manual* for complete information on each call.

This chapter does not describe all the Paragon system's system calls. For information about system calls that provide general services other than message passing, see Chapter 4. For information about the calls used with the Parallel File System, see Chapter 5. For information about the calls used with graphical interfaces, such as DGL and the X Window System, see the *Paragon™ System Graphics Libraries User's Guide.* For information about the system calls that require root privileges, see the *Paragon™ System Administrator's Guide.*

Programs written in C can also issue OSF/1 system calls. The operating system is a complete OSF/1 operating system and fully supports all the standard OSF/1 system calls. See the *OSF/1 Programmer's Reference* for information on these calls.

Programs written in Fortran cannot make OSF/1 system calls directly, but the Fortran runtime library includes a number of system interface routines. These routines make a number of OSF/1 system calls available to Fortran programs. See the *Paragon™ System Fortran Compiler User's Guide* for information on these routines.

Process Characteristics

Each process within an application is identified by its *node number* and *process type*. A process must have a valid node number and process type to send and receive messages.

Node Numbers

Synopsis	Description
mynode()	Obtain the calling process's node number.
numnodes()	Obtain the number of nodes allocated to the current application.

A process's *node number* is an integer that identifies the node on which it is running. Node numbers are assigned by the system, and range from zero to one less than the number of nodes in the application. A process can find out its node number by calling **mynode()**; the node number does not change for the life of the process. A process can also find out the number of nodes in the application by calling **numnodes()**; the maximum node number in the application is **numnodes() – 1**.

When you run an application that was linked with the **-nx** switch, the system creates one process on each node of the default partition (unless you specify otherwise on the application's command line). Each process is the same as the others except for its node number, which is different in each process.

All message-sending system calls have a *node* parameter that specifies the node to which the message is sent. You can use any valid node number, or the special value -1 to send the message to all nodes in the application except the sending node itself.

Some message-receiving system calls have a *nodesel* parameter that specifies the node from which the message was sent. A *nodesel* parameter can be a valid node number (to receive only messages from that node), or the special value -1 (to receive messages from any node). Message-receiving system calls that do not have a *nodesel* parameter always receive messages from any node.

The node numbers used in message-passing calls are always node numbers within the application, not physical slot numbers or node numbers within the partition in which the application is running. For example, if you run an application on 30 nodes of a 64-node partition by using the switch **-sz 30**, the node numbers within the application will always be 0 through 29. However, those nodes might not be nodes 0 through 29 of the partition. They might be nodes 0 through 29, or 10 through 39, or a completely arbitrary set of nodes.

Process Types

Synopsis	Description
myptype()	Obtain the calling process's process type.
setptype(*ptype*)	Set the calling process's process type (only permitted if the process type is currently **INVALID_PTYPE**).

A process's *process type*, or *ptype*, is an integer that distinguishes the process from other processes in the same application running on the same node. Process types are assigned by the user, and can be any integer from 0 to 2,147,483,647 ($2^{31} - 1$) inclusive. A process can find out its process type by calling **myptype()**. A process cannot change its process type once it has been set to a valid value.

When you run an application that was linked with **-nx**, the system sets the process type of all processes in the application to the value you specify with the **-pt** switch on the application's command line (default 0).

All message-sending system calls have a *ptype* parameter that specifies the process type to which the message is sent. You must specify the process type; you cannot use -1.

Some message-receiving system calls have a *ptypesel* parameter that specifies the process type from which the message was sent. A *ptypesel* parameter can be a valid process type (to receive only messages from that process type), or the special value -1 (to receive messages from any process type). Message-receiving system calls that do not have a *ptypesel* parameter always receive messages from any process type.

Certain system calls that involve all the nodes in the application, called *global operations*, require that every node in the application has one process with the same process type. All these processes must call the global operation before the application can proceed.

Within a single application, multiple processes running on the same node must have different process types. However, processes on different nodes may (and usually do) have the same process type. Two processes running on a single node may have the same process type only if they belong to different applications.

NOTE

The **-pt** switch (or, if not specified, the default process type of 0) applies only to the process type of the initial processes created by running the application.

If an application creates additional processes after it starts up, and no process type is specified for the new process, the new process's process type is set to the special value **INVALID_PTYPE** (a negative constant defined in the header file *nx.h*). A process whose process type is **INVALID_PTYPE** cannot send or receive messages. It must use the system call **setptype()** to set its process type to a valid value before it can send or receive any messages. (This is the only valid use of **setptype()**.)

The system calls that create node processes (**nx_nfork()**, **nx_load()**, and **nx_loadve()**) have a *ptype* parameter that specifies the process type of the newly-created processes. However, the standard OSF/1 system call **fork()**, which creates a new process on the same node as the process that calls it, does not provide any way to specify the new process's process type. This means that the process type of a process created by **fork()** is set to **INVALID_PTYPE**. The new process must call **setptype()** before it can send or receive messages. The specified process type must be different from the parent's, and different from the process type of any other process in the same application on the same node.

A process's process type is inherited across an **exec()**. This means that if you do a **fork()** followed by an **exec()**, you can call **setptype()** either before or after the **exec()**. However, the **setptype()** must follow the **fork()**.

Once a process has used a process type, that process type is associated with the process for the life of the application. No other process on the same node in the same application can ever use that process type, even if the original process terminates.

If a process has multiple pthreads, all the pthreads in the process have the same process type. See Chapter 6 for information on pthreads.

Message Characteristics

Messages are characterized by a *length*, a *type*, and sometimes an *ID*. These characteristics are set when the message is sent, and do not change for the life of the message.

Message Length

The *length* of a message is the number of bytes of information contained in the message. Messages can be of any length.

All message-passing system calls have a *count* parameter that specifies the length of the message to be sent or received. The length you specify must be less than or equal to the size in bytes of the buffer used in the call. Message-sending calls read exactly that number of bytes from the buffer and send them as a message; message-receiving calls generate an error if a message is received that is larger than the specified length.

If you program in C, when you send a message you can use the **sizeof** operator to determine the size of your message in bytes. If you program in Fortran, you will need to add up the sizes of all the data elements within the message; see the *Paragon™ System Fortran Compiler User's Guide* for information on the default size of each data type. If you pass named common blocks as messages, you may also have to include the space taken up by padding within the common block, as discussed under "Message Passing with Fortran Commons" on page 3-17.

You can also send and receive zero-length messages. This is useful if the message type is sufficient, and there is no need to supply any message content. For example, one process could tell another process to start or stop doing something by sending a zero-length message of type 1 to start, or a zero-length message of type 2 to stop.

Message Type

The *type* of a message is an integer whose meaning is determined by the programmer.

All message-sending system calls have a *type* parameter that specifies the type of the message sent. You can use any integer from 0 to 999,999,999 (inclusive) as a message type.

All message-receiving system calls have a *typesel* parameter that specifies the type (or types) of messages the call will receive. A *typesel* parameter can be an integer from 0 to 999,999,999 (to receive only messages of the specified type) or the special value -1 (to receive messages of any type).

There are also special message types outside the range 0 to 999,999,999, called *force types* and *typesel masks*, that you can use. Sending with a force type sends a message that uses a limited flow control technique; receiving with a typesel mask receives messages of a selected set of types. See the *Paragon™ System Fortran Calls Reference Manual* or *Paragon™ System C Calls Reference Manual* for information on these special message types. Note, though, that regular messages are just as fast as force type messages, so force types are not needed for performance.

Message ID

The *ID* of a message is an identifier used to check for the completion of asynchronous messages. Synchronous messages do not have IDs.

When you send or receive a message with an *asynchronous* message-passing call (one that returns before the message is completely sent or received), the call returns an ID that you can use to check whether or not the send or receive is complete. See "Asynchronous Send and Receive" on page 3-10 for more information on message IDs.

Message Order

The operating system guarantees that all messages will arrive in the same order they are sent. That is, if one message is sent from node A to node B, then a second message is sent from node A to node B, the second message will never arrive before the first.

Although the first message always *arrives* at the node first, you can elect to *receive* the second message—that is, to copy its contents into a buffer in user memory—before the first. You do this by specifying different message types in the send calls on node A, and specifying the second message's type in the first receive call on node B.

Names of Send and Receive Calls

You can tell what each message-passing call does by examining its name.

The first character of the name indicates whether the call is synchronous, asynchronous, or handled:

c Synchronous (complete) call. These calls do not return until the message is complete. They are discussed under "Synchronous Send and Receive" on page 3-8.

i Asynchronous (incomplete) call. These calls return immediately, so your program can do other work while the message is processed. They are discussed under "Asynchronous Send and Receive" on page 3-10.

h Asynchronous with interrupt handler (handled) call. Like the **i...()** calls, the **h...()** calls return immediately. Unlike the **i...()** calls, **h...()** calls indicate that the message is complete by calling a user-supplied interrupt handler. They are discussed under "Treating a Message as an Interrupt" on page 3-18.

The initial **c**, **i**, or **h** is followed by a verb that indicates what the call does:

send	Send a message.
recv	Receive a message.
sendrecv	Send a message and receive the reply.
probe	Probe for a pending (not yet received) message.

Finally, the verb may be followed by an **x** to indicate that it is an "extended" version of the call (see "Treating a Message as an Interrupt" on page 3-18 and "Extended Receive and Probe" on page 3-24).

The synchronous calls with no additional functionality, such as **csend()**, are the easiest to understand and use. However, the asynchronous calls (such as **isend()**) and the calls with additional functionality (such as **crecvx()**) can offer dramatic improvements in performance when properly used.

Synchronous Send and Receive

Synopsis	**Description**
csend(*type*, *buf*, *count*, *node*, *ptype*)	Send a message, waiting for completion.
crecv(*typesel*, *buf*, *count*)	Receive a message, waiting for completion.
csendrecv(*type*, *sbuf*, *scount*, *node*, *ptype*, *typesel*, *rbuf*, *rcount*)	Send a message and post a receive for the reply. Wait for completion.

The **c...()** message-passing calls perform synchronous sends and receives.

- A synchronous send means that the program executing the send waits until the send is complete. This waiting is referred to as *blocking*. Completing the send, however, does not guarantee that the message has been received. It only means that the message has left the sending process and that the buffer can be reused. You use **csend()** to perform a synchronous send.

- A synchronous receive means that the program executing the receive waits until the message arrives in the specified buffer. You use **crecv()** to perform a synchronous receive.

- A **csendrecv()** is like a **csend()** followed by a **crecv()**. It returns the length of the received message.

Here are two code fragments in C that perform a synchronous send and a synchronous receive.

- Node 1 sends a message of type 0 to the process with the same process type on node 0:

```
#include <nx.h>
#define MSG_TYPE 0
#define DEST_NODE 0
char send_buf[100];
   •
   •
   •
csend(MSG_TYPE, send_buf,
   sizeof(send_buf), DEST_NODE, myptype());
```

- Node 0 receives the message:

```
#include <nx.h>
#define MSG_TYPE 0
char recv_buf[100];
    •
    •
    •
crecv(MSG_TYPE, recv_buf, sizeof(recv_buf));
```

See "Extended Receive and Probe" on page 3-24 for information on a version of the **crecv()** call with additional functionality.

Synchronous Send to Multiple Nodes

Synopsis	**Description**
gsendx(*type*, *buf*, *count*, *nodes*, *nodecount*)	Send a message to a list of nodes, waiting for completion.

The **gsendx()** call sends a message to multiple nodes. Specifically, it performs a synchronous send of the message specified by the *type*, *buf*, and *count* arguments to the process with the same process type as the caller on the nodes specified by the *nodes* argument. The *nodes* argument is an array of integers; the *nodecount* argument specifies the number of nodes in *nodes*.

For example, the following code fragment in Fortran sends the data in the array *x* to nodes 1 and 3:

```
integer*4 nodenums(2), x(10)
    •
    •
    •
nodenums(1) = 1
nodenums(2) = 3
call gsendx(100, x, 10*4, nodenums, 2)
```

Asynchronous Send and Receive

Synopsis	Description
isend(*type*, *buf*, *count*, *node*, *ptype*)	Send a message without waiting for completion.
irecv(*typesel*, *buf*, *count*)	Receive a message without waiting for completion.
isendrecv(*type*, *sbuf*, *scount*, *node*, *ptype*, *typesel*, *rbuf*, *rcount*)	Send a message and post a receive for the reply without waiting for completion.
msgdone(*mid*)	Determine whether a send or receive operation has completed.
msgwait(*mid*)	Wait for completion of a send or receive operation.
msgignore(*mid*)	Release a message ID as soon as a send or receive operation completes.

The **i...()** message-passing calls perform asynchronous sends and receives. The **msgdone()** and **msgwait()** calls are used with the **i...()** calls to determine when the message has completed; the **msgignore()** call is used to discard a message ID as soon as the message has completed.

Unlike a synchronous send or receive, an asynchronous send or receive does not block. It returns a unique message ID, which is not reused until released. You can use this ID to check for completion at a later time.

NOTE

The number of message IDs is limited, so you *must* release each ID after you use it. See "Releasing Message IDs" on page 3-12 for information on releasing message IDs.

You use **isend()** to perform an asynchronous send, and **irecv()** to perform an asynchronous receive. An **isendrecv()** is like an **isend()** followed by an **irecv()**, except that it returns only one message ID (for the receive). Asynchronous sends can be used together with synchronous receives, and vice versa. For example, a message sent by **isend()** could be received by **crecv()**.

You must make sure that an asynchronous operation has completed before you change the contents of the send buffer or use the contents of the receive buffer. To check if an asynchronous operation has completed, use the **msgdone()** call. It returns 1 if an asynchronous call has completed and 0 otherwise. To block until an asynchronous operation has completed, use the **msgwait()** call. Both **msgdone()** and **msgwait()** take the message ID as an input parameter.

The message ID belonging to an asynchronous receive is distinct from the message ID belonging to any companion asynchronous send. For example, if node 0 sends a message with **isend()** and node 1 receives the message with **irecv()**, the **isend()** has a different message ID from the **irecv()**. When the **isend()** completes, this does *not* indicate that the corresponding **irecv()** has completed.

For example, assume that your application knows that it's going to need a message up ahead. So it posts an asynchronous receive with **irecv()**. It then does work that does not require the message, believing that by the time it needs the message, it will have arrived. When the program comes to where it needs the message, it issues a **msgwait()**. If the message has in fact arrived, the **msgwait()** returns immediately. Otherwise, it blocks until the message arrives. Here is a Fortran code fragment that implements this technique.

Node 1 does an asynchronous send:

```
      include 'fnx.h'

      integer result, msg_sid
      integer MSG_TYPE, DEST_NODE
      double precision send_buf(100)
      parameter (MSG_TYPE = 1)
      parameter (DEST_NODE = 0)
         •
         •
         •
      msg_sid = isend(MSG_TYPE, send_buf,
         100*8, DEST_NODE, myptype())
         •
         •
         •
c     Free the asynchronous send ID
      call msgwait(msg_sid)
```

Node 0 does the asynchronous receive:

```
         include 'fnx.h'

         integer result, msg_rid
         integer MSG_TYPE
         double precision rec_buffer(100)
         parameter (MSG_TYPE = 1)
             •
             •
             •
   c Post the receive
         msg_rid = irecv(MSG_TYPE, rec_buffer, 100*8)
             •
             •
             •
   c Now you need the message.
   c
   c Free the asynchronous receive ID
         call msgwait(msg_rid)
```

When the **msgwait()** returns, the message has been received. You may have blocked on the **msgwait()** if the message had not yet arrived. You may now assign another value to *msg_rid.*

See "Extended Receive and Probe" on page 3-24 for information on a version of the **irecv()** call with additional functionality.

Releasing Message IDs

Because the operating system has a limited number of message IDs, you must release IDs that are no longer needed. There are three ways to release a message ID:

- You can call **msgwait()**.
- You can keep calling **msgdone()** until it returns 1.
- You can call **msgignore()**.

If you use **msgignore()**, it tells the system to release the message ID as soon as the corresponding send or receive has completed. Note, though, that this leaves you with no way to determine whether or not the message has completed. In this case, your application must have some other means of synchronization to prevent the send or receive buffer from being used before the message is complete.

NOTE

Re-using a send or receive buffer before the message is complete can result in unexpected behavior. Do not use **msgignore()** unless you are certain this will not occur.

Merging Message IDs

Synopsis	Description
msgmerge(*mid1*, *mid2*)	Merge two message IDs into a single ID that can be used to wait for completion of both operations.

The **msgmerge()** call gives you a way to merge two or more message IDs together. It takes two message IDs as parameters, and returns a message ID that does not complete until both the messages identified by the input message IDs have completed.

Once you have merged a message ID with **msgmerge()**, you should not use the input message IDs as arguments to **msgwait()**, **msgdone()**, **msgcancel()**, or **msgignore()**. The input message IDs are automatically released when the merged message IDs are waited for.

For example, the following C code fragment posts two **irecv()**s, one for a message of type 1 and the other for a message of type 2, and then waits until both have completed:

```
#include <nx.h>

int mid1, mid2, midg;
char buf1[10], buf2[10];
   •
   •
   •
mid1 = irecv(1, buf1, 10);
mid2 = irecv(2, buf2, 10);

midg = msgmerge(mid1, mid2);

msgwait(midg);
```

Note that *mid1* and *mid2* are released by the **msgwait()** call on *midg*.

You can use a series of **msgmerge()** calls to merge multiple message IDs together. To help you do this, you can use the value -1 as one of the message IDs; **msgmerge()** returns the other message ID unchanged.

For example, the following Fortran code fragment uses a series of **isend()** calls to send the buffer *buf* as a message of type 1 to the process with the same process type on nodes 1 through 10, then waits for all of the **isend()**s to complete:

```
include 'fnx.h'

integer i, mid
integer buf(100)

mid = -1
i = 1

do while (i .le. 10)
   mid = msgmerge(mid, isend(1, buf, 400, i, myptype()))
   i = i + 1
end do

call msgwait(mid)
```

The message ID returned by each **isend()** call is merged together with the message IDs of the previous **isend()** calls into the merged message ID *mid* (the first message ID is merged with -1, yielding itself). Once all the **isend()**s have been posted, the program uses **msgwait()** on the merged message ID to wait for all of the **isend()**s to complete.

Probing for Pending Messages

Synopsis	**Description**
cprobe(*typesel***)**	Wait for a message of a selected type to arrive.
iprobe(*typesel***)**	Determine whether a message of a selected type is pending.

When a message arrives for which no receive has been issued, it goes into a system buffer. It is referred to as a *pending message*: a message that is available for receipt, but not yet received. When you issue a receive for that message, the message is moved into the application's buffer (the buffer you specify in the **crecv()** or **irecv()** call). If a receive has already been issued when the message arrives, it goes directly into the application's buffer and bypasses the system buffer.

The **cprobe()** and **iprobe()** calls determine whether there is a message of a given type pending in the system buffer. You can use a message type from 0 to 999,999,999 to probe for a message of a specific type; the special value -1 to probe for a message of any type; or a *typesel mask* to probe for messages of a selected set of types (see the *Paragon™ System Fortran Calls Reference Manual* or *Paragon™ System C Calls Reference Manual* for information on typesel masks).

The **cprobe()** call is a blocking call. It takes a type selection parameter as input and returns when a message of the given type has arrived. The **iprobe()** call is similar to **cprobe()**, except that it is nonblocking. **iprobe()** returns 1 if the message is pending and 0 if it is not.

cprobe() and **iprobe()** are not the only calls that probe for messages. See "Extended Receive and Probe" on page 3-24 for information on message-probing calls with additional functionality.

Getting Information About Pending or Received Messages

Synopsis	Description
infocount()	Return size in bytes of a pending or received message.
infonode()	Return node number of the node that sent a pending or received message.
infoptype()	Return process type of the process that sent a pending or received message.
infotype()	Return message type of a pending or received message.

The **info...()** calls return information about received or pending messages. You can obtain the size of the message, its type, and the node number and process type of the sending process.

The return value of the **info...()** calls is defined only in the following cases:

- After a **crecv()**, **cprobe()**, or **msgwait()**.
- After an **iprobe()** or **msgdone()** returns 1.

Note that you must issue the **info...()** call before you perform any other message-passing operation. Otherwise, you will get information about the most recently received or pending message instead.

For example, the following C code receives a message of any type, then uses **infotype()** to determine what type of message was actually received:

```
#include <nx.h>
#define BIGNUM 262144
long buf[BIGNUM], msg_type;
    •
    •
    •
crecv(-1, buf, sizeof(buf));
msg_type = infotype();
```

Another example: the following C code blocks until any message arrives, then allocates a buffer just large enough to hold the message and receives it:

```
#include <nx.h>
char *buf;
long msg_type, msg_len;
    •
    •
    •
cprobe(-1);
msg_type = infotype();
msg_len = infocount();
buf = (char *) calloc(msg_len, 1);
crecv(msg_type, buf, msg_len);
    •
    •
    •
```

Between the **cprobe()** and the **crecv()**, the message is pending; it has arrived, but has not yet been received. Until the message is received, the contents of the message are not accessible to the program.

The **info...()** calls are subject to the following special conditions:

- The return value of the **info...()** calls is undefined after a **msgwait()** or **msgdone()** if the message ID in the **msgwait()** or **msgdone()** call is a "merged" message ID representing more than one message. See "Merging Message IDs" on page 3-13 for more information.

- The return value of the **info...()** calls is undefined after a **crecvx()**, **cprobex()**, or **iprobex()**, except if the last parameter is the special array *msginfo*. See "Extended Receive and Probe" on page 3-24 for more information.

- If you issue an **info...()** call before doing *any* message passing, the call returns -1.

The **info...()** calls are not the only way to get information about a received or pending message. See "Extended Receive and Probe" on page 3-24 for information on message-receiving and message-probing calls that also return information about the received or pending message.

Message Passing with Fortran Commons

Fortran users often use common blocks to send messages that contain data elements of different types. For example, consider the named common containing a double precision number and an integer. It is good Fortran practice to put the largest data element first in the common list, as follows:

```
integer i
double precision d
common/msg/ d, i
```

To send this common block, specify the name of the first common element as the buffer and the length of the entire common as the length. For example, to send the common block named *msg*, send the variable *d* with a length of 12 bytes (8 for the double precision variable plus 4 for the integer variable). The following **csend()** call sends *msg* to process *ptype* on node *node*.

```
call csend(MSGTYPE, d, 12, node, ptype)
```

If you put smaller data elements before larger data elements in common blocks, the compiler may have to insert padding, or "holes," between the elements of the common block to preserve data alignment. For example, if you define the common block named *pmsg* as follows, the compiler will place an invisible 4-byte pad between the end of *i* and the beginning of *d* to properly align *d* on an 8-byte boundary:

```
integer i
double precision d
common/pmsg/ i, d
```

This padding has two effects:

- If you send this common block as a message, you must include the padding in the length of the message. For example, even though *pmsg* contains the same two variables as *msg*, *pmsg* is 4 bytes longer than *msg* because of the padding between *i* and *d*. To send *pmsg* to process *ptype* on node *node*, you would use the following call:

  ```
  call csend(MSGTYPE, i, 16, node, ptype)
  ```

- If another routine uses a different view of the same common block, you may have to add additional variables to the other routine's declaration of the common block to take this padding into account. For example, if another routine wants to view *d* in *pmsg* as an array of two integers, it must declare *pmsg* as follows:

  ```
  integer i, ipad, id(2)
  common/pmsg/ i, ipad, id(2)
  ```

  The variable *ipad* corresponds to the 4-byte pad in the original routine's declaration of *pmsg*. Without this variable, the position of *id* would not correspond to the position of *d* in the original common block. This variable is necessary if *pmsg* is shared between these two routines, whether or not the two routines run on different nodes.

When possible, you should define common blocks with the largest data element first, to avoid padding completely. You should also use the **%LOC** function to determine the size of a common block and avoid specifying its size with a hard-coded constant.

Treating a Message as an Interrupt

Synopsis	Description
hsend(*type*, *buf*, *count*, *node*, *ptype*, *handler*)	Send a message and set up a handler procedure to be called when the send completes.
hrecv(*typesel*, *buf*, *count*, *handler*)	Receive a message and set up a handler procedure to be called when the receive completes.
hsendrecv(*type*, *sbuf*, *scount*, *node*, *ptype*, *typesel*, *rbuf*, *rcount*, *handler*)	Send a message and post a receive for the reply. Set up a handler procedure to be called when the reply arrives.

The **h...()** message-passing calls perform asynchronous sends and receives. However, unlike the **i...()** calls, the **h...()** calls let you establish a user-provided interrupt handler, which is called when the send or receive is complete.

The **h...()** receive calls let you treat incoming messages as interrupts. For example, consider a program that performs some action based on the type of a received message. One way to implement this program is to block the program at a **crecv()** for messages of all types and then take appropriate action based on the value returned by **infotype()**.

Another way is to issue a number of **hrecv()** calls. Each call attaches a function to a particular message type or set of types. Your program does not block. You can continue with other work; but when the appropriate message comes, the attached function is called to take care of the message. (The message is stored in the receive buffer before the function is called.)

The handler function you define must be written in C and must have four arguments of type **long**. These arguments are passed the following values when the function is called:

1. Type of the message (as returned by **infotype()**).
2. Length of the message in bytes (as returned by **infocount()**).
3. Node number of the process that sent the message (as returned by **infonode()**).
4. Process type of the process that sent the message (as returned by **infoptype()**).

For example, here's a C code fragment that attaches the functions *funct0()*, *funct1()*, and *funct2()* to message types 0, 1, and 2, respectively. The message types that have handlers are referred to as *handled types*.

```
#include <nx.h>

char buf0[100], buf1[100], buf2[100];
void funct0(), funct1(), funct2();

hrecv(0, buf0, sizeof(buf0), funct0);
hrecv(1, buf1, sizeof(buf1), funct1);
hrecv(2, buf2, sizeof(buf2), funct2);
    •
    •   /* Now perform other work. No blocking happens. */
    •
```

The declaration of **funct1()** looks like this (the other functions are similar):

```
void funct1(long type, long count, long node, long ptype)
{
    •
    •
    •
}
```

When a message of type 1 arrives, the message is stored in the buffer specified in the **hrecv()** call (in this case, *buf1*), then **funct1()** is called with the type and length of the message and the node number and process type of the sender as arguments. **funct1()** and the main program then run concurrently until **funct1()** returns. (In previous releases of the operating system, the main program was interrupted and did not run at all until **funct1()** returned.)

CAUTION

The handler runs in the same memory space as the main program (but they have separate stacks).

Because of this, parts of the main program may have to be protected from being executed at the same time as the handler; see "Preventing Interrupts" on page 3-22 for information on using **masktrap()** to do this.

NOTE

Once you have established a handler for a message type, do not attempt to receive a message of that type with a **crecv...()** or **irecv...()** call.

hsend() operates the same as **hrecv()**, except that the handler is invoked when the send completes. (Note that completion of the send does not mean that the message has been received, only that the message has been sent and the send buffer can be reused.) **hsendrecv()** is like an **isend()** followed by an **hrecv()**, with the message ID of the **isend()** automatically released by **msgignore()**.

See "Extended Receive and Probe" on page 3-24 for information on a version of the **hrecv()** call with additional functionality.

Passing Information to the Handler

Synopsis	Description
hsendx(*type*, *buf*, *count*, *node*, *ptype*, *xhandler*, *hparam***)**	Send a message and set up an extended handler procedure to be called with the value *hparam* when the send completes. Allows handler sharing.

hsendx() is identical to **hsend()** except that it has an additional parameter, *hparam*, which is passed to the handler when it is called. The declaration of a handler for **hsendx()** looks like this:

```
void xhandler(long type, long count, long node, long ptype,
              long hparam)
{
    •
    •
    •
}
```

You can use the *hparam* parameter to write handlers that are shared among several **hsendx()** calls, each of which uses a different value of *hparam* to identify itself. For example, here is a C program fragment that sends two messages of type 0 to the process with process type 2 on node 1, then uses an **hsendx()** handler to free each message buffer as soon as the message send completes:

```
#include <nx.h>
#include <malloc.h>

#define NBUFS 2
#define BUFFER_SIZE 10000

char *buf[NBUFS]; /* array of pointers to char */

void freemem(long type, long count, long node, long ptype,
             long hparam)
{
    if( (hparam >= 0) && (hparam < NBUFS) ) {
        free(buf[hparam]);
    } else {
        printf("freemem(): invalid value: %d\n", hparam);
    }
}

main(int argc, char **argv)
{
    /* allocate two buffers with malloc() */
    buf[0] = malloc(BUFFER_SIZE);
    buf[1] = malloc(BUFFER_SIZE);
       •
       •   /* put data into the buffers */
       •
    /* send them */
    hsendx(0, buf[0], BUFFER_SIZE, 1, myptype(), freemem, 0);
    hsendx(0, buf[1], BUFFER_SIZE, 1, myptype(), freemem, 1);
       •
       •   /* Now perform other work */
       •
}
```

Note that you must take care that this handler is not called while the program is in the middle of a call to **malloc()** or **free()**. If the handler attempts to free memory while another part of the program is allocating or freeing memory, **malloc()**'s internal memory structures could become corrupted. You can prevent this by using the **masktrap()** call, described in the following section, to protect each **malloc()** and **free()** call elsewhere in the program that could be interrupted by this handler.

Preventing Interrupts

Synopsis	Description
masktrap(*state*)	Enable or disable interrupts for message handlers. Required to prevent corruption of global variables.

If you have one or more handlers set up and you have some critical code that you do not want interrupted, use the **masktrap()** call. A *state* value of 1 prevents any handler from running; a *state* value of 0 (zero) re-enables handlers. Any pending interrupts are honored when the mask is removed. A **masktrap()** call returns the previous state value (1 or 0). For example:

```
hrecv(6,buf,sizeof(buf),myhandler);
    •
    • /* this code can be interrupted */
    • /* by a message of type 6 */
    •
oldmask = masktrap(1);
    •
    • /* critical code that must not be interrupted */
    •
masktrap(oldmask);
    •
    • /* this code can be interrupted again */
    •
```

Note the use of the variable *oldmask* to save the value of the previous masking state before the call to **masktrap()**. This means that if the mask were already set before this call (for example, if this code is in a subroutine that could be called when the mask is already set), the following **masktrap()** call with the *oldmask* value as the argument would not unset it.

CAUTION

You must use **masktrap()** around any code in the main program that could interfere with calls in the handler.

For example, if the handler performs any I/O, you must put **masktrap()** calls around any I/O calls (such as **printf()**) in the main program that could be called while the handler is active. If you don't do this, you could find characters from the handler's output interleaved with characters from the main program's output.

Sometimes, it is not as obvious which calls could interfere with each other. For example, any two library calls that could allocate or free memory could cause the memory subsystem to become confused if they were called at the same time. To be on the safe side, keep the handler as simple as possible and use **masktrap()** to protect *all* library calls in the rest of the program that could call the same subsystems as the calls in the handler while the handler is active.

These calls to **masktrap()** are necessary because, when the handler is active, the handler and the main program share the same memory space and can change each other's global variables. This could cause any *non-reentrant* function to fail if it is called by both at the same time.

If the handler performs any message passing, any **info...()** call in the main program must be within the same set of **masktrap()** calls as the message-receiving call to which it applies. Otherwise, the **info...()** call could reflect the value of a message received within the handler.

NOTE

You do not have to use **masktrap()** in your main program to protect library calls that are in the standard C library (**libc.a**), if your application is linked with the reentrant C libraries (**-lpthreads** and **-lc_r**).

If you link your application with the reentrant C libraries, the standard C library is thread safe and you do not have to protect the calls that are in the standard C library. See "SMP Programming" on page 6-1 for information about the limitations of using reentrant C libraries in applications.

Extended Receive and Probe

Synopsis	Description
crecvx(*typesel*, *buf*, *count*, *nodesel*, *ptypesel*, *info*)	Receive a message of a specified type from a specified sending node and process type, together with information about the message. Wait for completion.
irecvx(*typesel*, *buf*, *count*, *nodesel*, *ptypesel*, *info*)	Receive a message of a specified type from a specified sending node and process type, together with information about the message. Do not wait for completion.
hrecvx(*typesel*, *buf*, *count*, *nodesel*, *ptypesel*, *xhandler*, *hparam*)	Receive a message of a specified type from a specified sending node and process type. Set up an extended handler procedure to be called with information about the message and the value *hparam* when the receive completes.
cprobex(*typesel*, *nodesel*, *ptypesel*, *info*)	Wait for a message of a specified type from a specified sending node and process type. Return information about the message.
iprobex(*typesel*, *nodesel*, *ptypesel*, *info*)	Determine whether a message of a specified type from a specified sending node and process type is pending. If it is, return information about the message.

The extended receive and probe calls, **crecvx()**, **irecvx()**, **hrecvx()**, **cprobex()**, and **iprobex()**, can be used to receive or probe for a message from a particular node or a particular process type, and return information about the message along with the message (instead of using the **info...()** calls).

crecvx(), **irecvx()**, **cprobex()**, and **iprobex()** are like **crecv()**, **irecv()**, **cprobe()**, and **iprobe()**, except that they have the following additional parameters:

nodesel Specifies the node that sent the message, or -1 for any node.

ptypesel Specifies the process type that sent the message, or -1 for any process type.

info An array of eight long integers that receives information about the specified message. The following information is stored into the first four elements of this array:

- Type of the message (as returned by **infotype()**).
- Length of the message in bytes (as returned by **infocount()**).
- Node number of the process that sent the message (as returned by **infonode()**).
- Process type of the process that sent the message (as returned by **infoptype()**).

The remaining four elements of the array are reserved.

hrecvx() is like **hrecv()**, except that it has the same *nodesel* and *ptypesel* parameters as the other **...x()** calls and the same *hparam* parameter as the **hsendx()** call. **hrecvx()** does not have an *info* parameter, because the corresponding information is passed to the handler when it is called.

The *info* parameter of **crecvx()**, **irecvx()**, **cprobex()**, and **iprobex()** must be specified and must not be zero or null. If you do not want this information, or you want it to be available to the **info...()** calls, specify the special array *msginfo*, defined in *nx.h* or *fnx.h*. The array *msginfo* is used by the non-x versions of these calls, and the **info...()** calls get their information from *msginfo*. This is why you cannot use the **info...()** calls after **crecvx()**, **cprobex()**, or **iprobex()** unless you specify *msginfo* as the last parameter of the extended receive or probe call.

The *info* parameter of **irecvx()** does not contain valid data until the message is received (as determined by **msgdone()** or **msgwait()**). The *info* parameter of **iprobex()** does not contain valid data unless the **iprobex()** returns 1.

For example, the following call receives a message of type 0 from process type 2 on node 1, storing information about the received message into the array *myinfo*:

```
/* C version */
char buf[80];
long myinfo[8];
crecvx(0, buf, sizeof(buf), 1, 2, myinfo);
```

After this **crecvx()** call, the message type is in *myinfo[0]*, its length is in *myinfo[1]*, the sender's node number is in *myinfo[2]*, and the sender's process type is in *myinfo[3]*.

```
c     Fortran version
      character*80 buf
      integer*4 myinfo(8)
      call crecvx(0, buf, len(buf), 1, 2, myinfo)
```

After this **crecvx()** call, the message type is in *myinfo(1)*, its length is in *myinfo(2)*, the sender's node number is in *myinfo(3)*, and the sender's process type is in *myinfo(4)*.

Note that the standard **crecv()** call

```
crecv(typesel, buf, count);
```

is exactly equivalent to the following **crecvx()** call:

```
crecvx(typesel, buf, count, -1, -1, msginfo);
```

Global Operations

Synopsis	Description
gcol(*x*, *xlen*, *y*, *ylen*, *ncnt*)	Concatenation.
gcolx(*x*, *xlens*, *y*)	Concatenation for contributions of known length.
gdhigh(*x*, *n*, *work*)	Vector double precision MAX.
gdlow(*x*, *n*, *work*)	Vector double precision MIN.
gdprod(*x*, *n*, *work*)	Vector double precision MULTIPLY.
gdsum(*x*, *n*, *work*)	Vector double precision SUM.
giand(*x*, *n*, *work*)	Vector integer bitwise AND.
gihigh(*x*, *n*, *work*)	Vector integer MAX.
gilow(*x*, *n*, *work*)	Vector integer MIN.
gior(*x*, *n*, *work*)	Vector integer bitwise OR.
giprod(*x*, *n*, *work*)	Vector integer MULTIPLY.
gisum(*x*, *n*, *work*)	Vector integer SUM.
gland(*x*, *n*, *work*)	Vector logical AND.
glor(*x*, *n*, *work*)	Vector logical inclusive OR.
gopf(*x*, *xlen*, *work*, *function*)	Arbitrary commutative function.
gshigh(*x*, *n*, *work*)	Vector real MAX.
gslow(*x*, *n*, *work*)	Vector real MIN.
gsprod(*x*, *n*, *work*)	Vector real MULTIPLY.
gssum(*x*, *n*, *work*)	Vector real SUM.
gsync()	Global synchronization.

The **g...()** calls perform operations that use data from every node in the application. In general, when you make one of these calls, each node contributes a piece of data to the operation, the operation is performed on the whole collection of data, and then the result of the operation is returned to each node.

These operations are *synchronizing calls*: if any node in an application makes one of these calls, it blocks until every node in the application has made the same call. (In the simplest case, **gsync()**, this synchronization is the *only* operation performed by the call.) One process on each node in the application must make the call, and all the processes that make the call must have the same process type.

The global operations are implemented using dynamic algorithm selection for maximum performance. The system considers several ways of exchanging the needed information between the nodes, and selects the one that minimizes the time required to perform the global operation given the size and shape of the application.

Each global operation's name begins with **g** and ends with the name of the operation. Some operations have several versions, which operate on different data types; for these calls, the data type is indicated by the second letter of the call's name (**l** for logical, **i** for integer, **s** for single-precision floating point, or **d** for double-precision floating point). For example, **gdsum()** performs a **g**lobal **d**ouble-precision **sum**.

To illustrate the use of a global operation, consider the **gdsum()** call. This call is used by the π example discussed under "Example Application: Calculating pi" on page 7-7. This example evaluates π by calculating a definite integral. The integral is partitioned among the nodes of a cube. The answer, then, is the sum of the answers from each of the participating nodes. Here's a code fragment from the Fortran version of the example:

```
double precision pi,tmp
    •
    •
    •
call gdsum(pi,1,tmp)
```

Before this **gdsum()** call, this node's part of the total integral is stored in the variable *pi*. **gdsum()** is designed to operate on a vector, so the second parameter specifies the size of the vector; in this case, it is a "vector" of size 1 (a single variable). The third parameter, *tmp*, is a temporary area used in the calculation. After this **gdsum()** call, the sum of all the nodes' *pi*'s is stored in every node's *pi*.

Managing Applications and Partitions with System Calls

4

Introduction

This chapter describes the system calls that let you create and manage applications and partitions on Paragon systems. This chapter also describes the system calls that perform general services other than message passing. The following sections, each of which describes a group of related calls:

- Managing applications.
- Managing partitions.
- Listing unusable nodes.
- Handling errors.
- Controlling floating-point behavior.
- Miscellaneous calls.
- iPSC® system and Touchstone DELTA system compatibility calls.

Within each section, the calls are discussed in order of increasing complexity. That is, the "base" calls are discussed first, and the "extended" calls are discussed later.

Each section includes numerous examples in both C and Fortran. A call description at the beginning of each section or subsection gives a language-independent synopsis (call name, parameter names, and brief description) of each call discussed in that section. Differences between C and Fortran are noted where applicable. See Appendix A for information on call and parameter types; see the *Paragon™ System C Calls Reference Manual* or the *Paragon™ System Fortran Calls Reference Manual* for complete information on each call.

This chapter does not describe all the system calls. For information about system calls that perform message passing, see Chapter 3. For information about the calls used with the Parallel File System, see Chapter 5. For information about the calls used with graphical interfaces, such as DGL and the X Window System, see the *Paragon™ System Graphics Libraries User's Guide.* For information about the system calls that require root privileges, see the *Paragon™ System Administrator's Guide.*

Applications written in C can also issue OSF/1 system calls. The operating system is a complete OSF/1 system and fully supports all the standard OSF/1 system calls. See the *OSF/1 Programmer's Reference* for information on these calls.

Applications written in Fortran cannot make OSF/1 system calls directly, but the Fortran runtime library includes a number of system interface routines. These routines make a number of OSF/1 system calls available to Fortran programs. See the *Paragon™ System Fortran Compiler User's Guide* for information on these routines.

NOTE

Do not use the Mach system call interface.

This interface is not supported. It is not documented in SSD manuals, but you may read about Mach elsewhere. If you use Mach system calls, your application may fail. Mach memory allocation and Paragon memory allocation do not work together.

Managing Applications

The operating system provides system calls that let you create parallel applications, control their execution, and get information about them. See "Running Applications" on page 2-11 and "Managing Running Applications" on page 2-29 for introductory information on applications.

Controlling Application Execution with System Calls

Synopsis	Description
nx_initve(*partition*, *size*, *account*, *argc*, *argv*)	Create a new application.
nx_initve_rect(*partition*, *anchor*, *rows*, *cols*, *account*, *argc*, *argv*)	Create a new application with a rectangular shape.
nx_initve_attr(*partition*, *argc*, *argv*, [*attribute*, *value*,]... **NX_ATTR_END**)	Create a new application with specified attributes.
nx_pri(*pgroup*, *priority*)	Set the priority of an application.
nx_nfork(*node_list*, *numnodes*, *ptype*, *pid_list*)	Copy the current process onto some or all nodes of an application.
nx_load(*node_list*, *numnodes*, *ptype*, *pid_list*, *pathname*)	Execute a stored program on some or all nodes of an application.
nx_loadve(*node_list*, *numnodes*, *ptype*, *pid_list*, *pathname*, *argv*, *envp*)	Execute a stored program on some or all nodes of an application, with specified argument list and environment.
nx_waitall()	Wait for all application processes.

The simplest way to control the way an application executes is to use the command-line switch **-nx** when you link the application. (See "Compiling and Linking Applications" on page 2-5 for more information on the **-nx** switch.) When you execute a program that was linked with **-nx**, the program is automatically copied onto the specified number of nodes in the specified partition, runs, and then when all the nodes have finished you get your prompt back.

The code linked in by **-nx** reads the command line and environment variables, then performs the following actions for you (see "Controlling the Application's Execution Characteristics" on page 2-12 for more information):

- Creates a new, empty application in the partition specified by the **-pn** switch and on the nodes of that partition specified by the **-sz** or **-nd** switch. If **-pn** is not used, the partition is specified by *$NX_DFLT_PART*, or *.compute* if *$NX_DFLT_PART* is not set. If neither **-sz** nor **-nd** is used, the number of nodes is specified by *$NX_DFLT_SIZE*, or all nodes of the partition if *$NX_DFLT_SIZE* is not set.

- Sets the application's priority to the value specified by **-pri** (default 5).

- Loads and starts your program(s) on the nodes specified by **-on** (default all nodes of the application) with the process type specified by **-pt** (default 0).

The **nx_...()** system calls perform the same actions as those of the code linked in by **-nx**, but under program control instead of command-line control. Using these calls is more complicated than using **-nx**, but gives your program more flexibility and control.

NOTE

If you use any of the **nx_initve...()** calls, do *not* link the program with the **-nx** switch. Use the switch **-lnx** instead.

The switch **-lnx** links in the library *libnx.a*, which contains all the calls discussed in this manual, but does not link in the automatic start-up code linked in by **-nx**.

Creating an Application with nx_initve()

nx_initve() creates a new, empty application. The process that calls **nx_initve()** becomes the new application's *controlling process*; see "The Controlling Process" on page 4-26 for information on what this means.

The partition and size of the new application can be specified by parameters or by the command line; the priority and *msg_switches* are specified by the command line. If command-line switches are not used or the command line is ignored by specifying zero for *argc*, the application's execution characteristics default as discussed under "Controlling the Application's Execution Characteristics" on page 2-12 and the *msg_switches* default as discussed under "Message-Passing Configuration Switches" on page 8-18.

nx_initve() just allocates the specified number of nodes from the partition; it does not start any processes. (This allocation may or may not be exclusive, depending on the characteristics of the partition.) You must call **nx_nfork**(), **nx_load**(), or **nx_loadve**() to start processes in the new application. The nodes allocated to the application are automatically deallocated when all the processes in the application have terminated.

Another effect of **nx_initve**() is to make sure that the calling process is a *process group leader*. If the calling process is not already a process group leader, **nx_initve**() creates a new process group, removes the calling process from its current process group, and makes the calling process the new process group's leader. If you're not familiar with these terms, see "Process Groups" on page 4-27.

Finally, **nx_initve**() also initializes the data structures required by all the other calls described in this manual. In an application linked with **-nx**, the code linked in by **-nx** performs this initialization before the application starts up, so you can use these other calls anywhere in the application. In an application linked with **-lnx**, however, you must call **nx_initve**() before you can use *any* of the other calls described in this manual. If called before **nx_initve**(), these other calls will fail; the way they fail depends on the call (as described under "Handling Errors" on page 4-55). For example, if you call **csend**() before calling **nx_initve**(), the **csend**() prints an error message and terminates the calling process.

The parameters of **nx_initve()** have the following meanings:

partition

The relative or absolute partition pathname of the partition to run the application in, or a null string (" " or **NULL** in C, " " in Fortran) to use the default partition (the partition specified by *$NX_DFLT_PART*, or *.compute* if *$NX_DFLT_PART* is not set). The specified partition must exist and must give execute permission to the calling process.

If the user specifies a partition with the **-pn** switch on the command line, it overrides the value of the *partition* parameter (unless you set the *argc* parameter to 0, as described later in this section).

See "Partition Pathnames" on page 2-33 for more information on partition pathnames; see "Owner, Group, and Protection Modes" on page 2-38 for more information on partition permissions.

size

The size of the application (number of nodes to run the application on), or 0 to use the default size (the size specified by *$NX_DFLT_SIZE*, or all nodes of the partition if *$NX_DFLT_SIZE* is not set).

nx_initve() attempts to allocate a square group of nodes if it can. If this is not possible, it attempts to allocate a rectangular group of nodes that is either twice as wide as it is high or twice as high as it is wide. If this is not possible, it allocates any available nodes. In this case, nodes allocated to the application may not be contiguous.

If the user specifies the **-sz** or **-nd** switch on the command line, it overrides the value of the *size* parameter (unless you set the *argc* parameter to 0, as described later in this section).

account

In the future, this parameter will be used for accounting information. For now, it must be a null string (" " or **NULL** in C, " " in Fortran).

argc

In C, a pointer to an integer whose value is the number of arguments on the command line (counting the application name). If the value of this integer is 0, the command line is ignored. You can use a pointer to the *argc* parameter of **main()**, or you can construct the command line yourself.

In Fortran, this parameter is any nonzero value to search the command line, or 0 to ignore the command line.

argv

In C, a pointer to the command-line arguments, which may include arguments that specify application characteristics. You can use the *argv* parameter of **main()**, or you can construct the command line yourself.

In Fortran, **nx_initve()** gets the command line directly from the system, because Fortran programs don't have an *argv* parameter. This parameter is ignored; it should always be 0.

In either language, if any of the command-line arguments **-sz** *size*, **-sz** *h***X***w*, **-nd** *h***X***w***:***n*, **-pri** *priority*, **-pn** *partition*, **-nt** *nodetype*, **-rlx**, **-pkt** *packet_size*, **-mbf** *memory_buffer*, **-mex** *memory_export*, **-mea** *memory_each*, **-sth** *send_threshold*, **-sct** *send_count*, **-gth** *give_threshold*, or **-plk** is found in the command line:

- The appropriate application characteristic is set as specified by the argument.
- The argument is removed from *argv*.
- The variable pointed to by *argc* is decremented appropriately.

Any remaining arguments are moved to the beginning of *argv* for your program's use.

Note that the arguments **-pt** *type*, **-on** *nodelist*, and **\;** *application* are *not* recognized by **nx_initve()**. If you want your application to have the same user interface as an application linked with **-nx**, you must examine the argument list for these arguments and pass the appropriate values to **nx_load()** or **nx_loadve()** yourself.

nx_initve() returns the number of nodes in the application, or -1 if any error occurs.

For example, the following C call creates an application whose characteristics (partition, number of nodes, and so on) are determined by the user through command-line switches. If the user runs this program with no command-line switches, it runs on the user's default number of nodes in the user's default partition.

```
#include <nx.h>

main(int argc, char *argv[]) {
    int n;

    n = nx_initve("", 0, "", &argc, argv);
```

After this call, the variable *n* contains the number of nodes in the new application, or -1 if any error occurred. The variable *argc* contains the count of arguments that were not recognized and removed by **nx_initve()**, and the array *argv* contains pointers to those arguments.

The following Fortran call creates an application on 50 nodes of the partition *mypart*, ignoring any command-line switches provided by the user:

```
include 'fnx.h'
integer n

n = nx_initve("mypart", 50, "", 0, 0)
```

After this call, the variable *n* contains the number of nodes in the new application, or -1 if any error occurred.

The following restrictions apply to **nx_initve()**:

- A single process cannot call **nx_initve()** more than once.
- An application that calls **nx_initve()** cannot be linked with **-nx**. You must use **-lnx** instead.
- If your application uses any signal handlers, you must set them up *after* the call to **nx_initve()**. See **signal()** in the *OSF/1 Programmer's Reference* for more information on signal handlers.

The reason you cannot use **-nx** when you link an application that calls **nx_initve()** is that the code linked in by **-nx** calls **nx_initve()** itself, and **nx_initve()** can only be called once in an application. If you do use **-nx** when you link, your application's call to **nx_initve()** (actually the second call to **nx_initve()**) fails and returns -1.

Creating a Rectangular Application with nx_initve_rect()

nx_initve_rect() works exactly like **nx_initve()** except that it requires that the nodes allocated to the application form a rectangle with a particular height and width. Optionally, it can also specify the rectangle's location within the partition. The parameters of **nx_initve_rect()** are the same as those of **nx_initve()**, except that instead of the *size* parameter it has the following three parameters:

anchor The node number within the partition for the upper left corner of the rectangle, or -1 to allow the rectangle to be placed anywhere in the partition it will fit.

rows The height of the rectangle.

cols The width of the rectangle.

If the specified rectangle of nodes is not available, the **nx_initve_rect()** call fails and returns -1 (even if the equivalent number of nodes is available with a non-rectangular shape).

NOTE

All the restrictions and cautions in this manual that refer to **nx_initve()** also apply to **nx_initve_rect()**.

If the user specifies a size or shape with the **-sz** or **-nd** switch on the command line, it overrides the values of these three parameters (unless you set the *argc* parameter to 0). **nx_initve_rect()** never uses the environment variable *$NX_DFLT_SIZE*.

For example, the following Fortran call creates an application 8 nodes high and 8 nodes wide (unless otherwise specified by command-line switches) anywhere it will fit in the user's default partition:

```
include 'fnx.h'
integer n

n = nx_initve_rect("", -1, 8, 8, "", 1, 0)
```

The following C call creates an application 10 nodes high and 20 nodes wide whose upper left corner is node 0 (the upper left corner of the partition) in the partition *mypart*, ignoring any command-line switches provided by the user:

```
#include <nx.h>
int n, i;

i = 0;
n = nx_initve_rect("mypart", 0, 10, 20, "", &i, NULL);
```

After either of these calls, the variable *n* contains the number of nodes in the new application, or -1 if any error occurred.

Note that **nx_initve_rect()** will fail if the exact specified rectangle is not available. If you just want to find out the application's size and shape, rather than mandating a particular size and shape, you can use an ordinary **nx_initve()**, followed by a call to **nx_app_rect()** (discussed under "Finding an Application's Shape with nx_app_rect()" on page 4-22) to determine the height and width assigned by **nx_initve()**.

Controlling Application Attributes with nx_initve_attr()

When you call **nx_initve()** or **nx_initve_rect()**, you specify only the partition and the number of nodes or rectangle of nodes. All the other application attributes you can specify with switches on the application command line, such as its priority and packet size, cannot be specified in the call's arguments; they are always extracted from the command line (*argv* argument).

nx_initve_attr() works exactly like **nx_initve()** except that you can specify *all* the application attributes in the call's arguments. The parameters of **nx_initve_attr()** are as follows:

partition The partition to run the application in, as for **nx_initve()**.

argc The command-line argument count, as for **nx_initve()**.

argv The command-line arguments, as for **nx_initve()**.

[*attribute*, *value*,] ...

A series of zero or more argument pairs that specify the application's attributes. The *attribute* is one of the constants described in Table 4-1; the *value* is the value of the specified attribute. The type of the *value* argument is determined by the value of the preceding *attribute* argument.

NX_ATTR_END A constant that marks the end of the list of *attribute*, *value* pairs.

NOTE

If you call **nx_initve_attr()** in a Fortran subprogram, you must include *fnx.h* after the subprogran declaration and before the call. This is required for the call to recognise the pre-defined attribute constants (for example, **NX_ATTR_SZ**).

NOTE

All the restrictions and cautions in this manual that refer to **nx_initve()** also apply to **nx_initve_attr()**.

The following table describes the *attribute*s and *value*s you can use with **nx_initve_attr()**. The attribute constants, including **NX_ATTR_END**, are defined in <*nx.h*> or <*fnx.h*>.

Table 4-1. Attribute Constants for Use with nx_initve_attr() (1 of 3)

Attribute Constant	Type of Following Value (C / Fortran)	Equivalent Switch	Description
NX_ATTR_SZ	**long** **INTEGER**	**-sz** *size*	Specifies the size of the application, like the *size* argument of **nx_initve()** (see "Creating an Application with nx_initve()" on page 4-4).
NX_ATTR_RECT	**long *** **INTEGER(2)**	**-sz** *h***X***w*	Specifies a rectangle for the application, like the *rows* and *cols* arguments of **nx_initve_rect()** (see "Creating a Rectangular Application with nx_initve_rect()" on page 4-7). The *value* is an array of two integers (height first, width second). If **NX_ATTR_ANCHOR** is not specified, the system determines the rectangle's location within the partition.

Table 4-1. Attribute Constants for Use with nx_initve_attr() (2 of 3)

Attribute Constant	Type of Following Value (C / Fortran)	Equivalent Switch	Description
NX_ATTR_ANCHOR	**long** **INTEGER**	**-nd** *h***X***w***:***n*	Specifies the node number of the upper left corner of the rectangle, like the *anchor* argument of **nx_initve_rect()** (see "Creating a Rectangular Application with nx_initve_rect()" on page 4-7). If the specified node number is -1, the system determines the rectangle's location within the partition. **NX_ATTR_RECT** must also be specified (anywhere in the argument list) to give the height and width of the rectangle.
NX_ATTR_RELAXED	**long** **INTEGER**	**-rlx**	Specifies whether or not the requested number of nodes can be relaxed (see "Relaxing Application Size" on page 2-17 and "Using Node Attributes with a Relaxed Application Size" on page 2-28). The value 0 means all requested nodes must be available; the value 1 relaxes this requirement. If the user specifies the **-rlx** switch on the command line, it overrides the value 0. The value 1 cannot be overridden. **NX_ATTR_RELAXED** cannot be used with **NX_ATTR_RECT**, unless **NX_ATTR_ANCHOR** is also specified with a value other than -1.
NX_ATTR_SEL	**char *** **CHARACTER *(*)**	**-nt**	Specifies the node type for the application. The *value* is a character string whose value is a node type specifier (see "Specifying Node Attributes" on page 2-25). If **NX_ATTR_SZ** is specified or **NX_ATTR_RECT** is specified with **NX_ATTR_ANCHOR**, the given nodes must all be available and have the specified node type, unless **NX_ATTR_RELAXED** is specified.
NX_ATTR_PRI	**long** **INTEGER**	**-pri**	Sets the priorty for the application (see "Specifying Application Priority" on page 2-18).

Table 4-1. Attribute Constants for Use with nx_initve_attr() (3 of 3)

Attribute Constant	Type of Following Value (C / Fortran)	Equivalent Switch	Description
NX_ATTR_PKT	long INTEGER	-pkt	Sets the size of each message packet (see "Packetization" on page 8-16).
NX_ATTR_MBF	long INTEGER	-mbf	Sets the total amount of memory allocated to message buffers (see "System Message Buffers" on page 8-16).
NX_ATTR_MEX	long INTEGER	-mex	Sets the value of *memory_export* (see "Message-Passing Configuration Switches" on page 8-18).
NX_ATTR_MEA	long INTEGER	-mea	Sets the amount of memory allocated to buffering messages from each other node (see "System Message Buffers" on page 8-16).
NX_ATTR_NOC	long INTEGER	-noc	Sets the total number of other processes from which each process expects to receive messages (see "System Message Buffers" on page 8-16).
NX_ATTR_STH	long INTEGER	-sth	Sets the send threshold for sending multiple packets (see "System Message Buffers" on page 8-16).
NX_ATTR_SCT	long INTEGER	-sct	Sets the number of bytes to send right away when the available memory is above *send_threshold* (see "System Message Buffers" on page 8-16).
NX_ATTR_GTH	long INTEGER	-gth	Sets the threshold for the "give me more messages" message (see "System Message Buffers" on page 8-16).
NX_ATTR_PLK	long INTEGER	-plk	Specifies whether or not the data area of each process should be locked into memory (see "Process Locking" on page 8-15). The value 1 locks all processes into memory; the value 0 does not lock. If the user specifies the **-plk** switch on the command line, it overrides the value 0. The value 1 cannot be overridden.

For each *attribute* in the **nx_initve_attr()** call, if the user specifies the equivalent application switch on the command line, it overrides the value specified for the attribute in the call (unless you set the *argc* parameter to 0, as described for **nx_initve()**). The **-sz** or **-nd** switch can override **NX_ATTR_SZ**, **NX_ATTR_RECT**, and **NX_ATTR_ANCHOR**.

Each *attribute* can appear at most once in the argument list. The order of the *attribute*s in the argument list is not significant.

Specifying the Nodes Allocated to the Application

nx_initve_attr() provides the following ways to specify the nodes allocated to the application:

- Use **NX_ATTR_SZ** alone.

  Requests the specified number of nodes. If the *value* is 0 or -1, requests the number of nodes specified by *$NX_DFLT_SIZE*, or all the nodes of the partition if *$NX_DFLT_SIZE* is not set.

  NX_ATTR_SZ attempts to allocate a square group of nodes if it can. If this is not possible, it attempts to allocate a rectangular group of nodes that is either twice as wide as it is high or twice as high as it is wide. If this is not possible, it allocates any available nodes. In this case, nodes allocated to the application may not be contiguous.

- Use **NX_ATTR_RECT** alone.

  Requests a rectangle of nodes of the specified height and width. The system places the rectangle within the partition.

- Use both **NX_ATTR_RECT** and **NX_ATTR_ANCHOR**.

  Requests a rectangle of nodes of the specified height and width, whose upper left corner is located at the specified anchor node. **NX_ATTR_RECT** and **NX_ATTR_ANCHOR** can appear in any order in the argument list. If the value of **NX_ATTR_ANCHOR** is -1, the system places the rectangle within the partition..

- Use **NX_ATTR_SEL** alone.

  Requests all the nodes of the specified node type in the partition.

- Use **NX_ATTR_SEL** together with **NX_ATTR_SZ**, **NX_ATTR_RECT**, and/or **NX_ATTR_ANCHOR**.

  Requests the nodes specified by the **NX_ATTR_SZ**, **NX_ATTR_RECT**, and/or **NX_ATTR_ANCHOR**, all of which must have the node type specified by the **NX_ATTR_SEL**. See "Running an Application on a Particular Node Type" on page 2-23 for more information.

- Do not use **NX_ATTR_SEL**, **NX_ATTR_SZ**, **NX_ATTR_RECT**, or **NX_ATTR_ANCHOR**.

Requests the number of nodes specified by *$NX_DFLT_SIZE*, or all the nodes of the partition if *$NX_DFLT_SIZE* is not set.

- Use **NX_ATTR_RELAXED** with a *value* of 1 together with any of the above.

  Requests all the *available* nodes in the specified node set, *up to* the number of nodes requested. At least one of the specified nodes must be available. See "Relaxing Application Size" on page 2-17 for more information.

  If **NX_ATTR_SEL** is used together with **NX_ATTR_RELAXED**, only nodes of the specified type are returned, but the number of nodes returned may be less than the number of nodes requested. At least one node of the specified type must be available. See "Using Node Attributes with a Relaxed Application Size" on page 2-28 for more information.

All these attributes can be overridden by command-line switches. In particular, note that either the **-sz** or **-nd** switch overrides **NX_ATTR_SZ**, **NX_ATTR_RECT**, and **NX_ATTR_ANCHOR**. If an attribute is overridden by a command-line switch, the effect is as though it had been specified in the **nx_initve_attr**() call with the value from the command line.

The following combinations of these attributes are invalid:

- You cannot use **NX_ATTR_ANCHOR** without **NX_ATTR_RECT**.

- You cannot use **NX_ATTR_SZ** together with **NX_ATTR_RECT**.

- You cannot use **NX_ATTR_RELAXED** together with **NX_ATTR_RECT**, unless **NX_ATTR_ANCHOR** is also specified with a value other than -1.

Using any of these combinations of attributes causes **nx_initve_attr()** to fail with the error "invalid attribute specified."

Examples

The following C call creates an application whose characteristics (partition, number of nodes, and so on) are determined by the user through command-line switches. If the user runs this program with no command-line switches, it runs on the user's default number of nodes in the user's default partition.

```
#include <nx.h>

main(int argc, char *argv[]) {
    int n;

    n = nx_initve_attr("", &argc, argv, NX_ATTR_END);
```

After this call, the variable *n* contains the number of nodes in the new application, or -1 if any error occurred. The variable *argc* contains the count of arguments that were not recognized and removed by **nx_initve()**, and the array *argv* contains pointers to those arguments.

The following Fortran call creates an application on 50 nodes of the partition *mypart*, ignoring any command-line switches provided by the user:

```
         include 'fnx.h'
         integer n

         n = nx_initve_attr("mypart", 0, 0,
        x                   NX_ATTR_SZ, 50,
        x                   NX_ATTR_END)
```

The following C call creates an application that consists of all available nodes in a rectangle 10 nodes high and 20 nodes wide whose upper left corner is node 0 (the upper left corner of the partition) in the partition *mypart*, ignoring any command-line switches provided by the user:

```
#include <nx.h>
long rect[2];
int i, n;

rect[0] = 10;
rect[1] = 20;
i = 0;

n = nx_initve_attr("mypart", &i, NULL,
                   NX_ATTR_RELAXED, 1,
                   NX_ATTR_RECT, rect,
                   NX_ATTR_ANCHOR, 0,
                   NX_ATTR_END);
```

The following Fortran call creates an application consisting of any 15 nodes in the user's default partition that are not I/O nodes and have exactly 32M bytes of RAM (unless otherwise specified by command-line switches):

```
         include 'fnx.h'
         integer n

         n = nx_initve_attr("", 1, 0,
        x                   NX_ATTR_SZ, 15,
        x                   NX_ATTR_SEL, "32mb, !io",
        x                   NX_ATTR_END)
```

After any of these calls, the variable *n* contains the number of nodes in the new application, or -1 if any error occurred.

Setting an Application's Priority with nx_pri()

nx_pri() sets the specified application's priority to the specified value. If you don't call **nx_pri()** and the user doesn't use the **-pri** switch, the default priority is 5. The parameters of **nx_pri()** have the following meanings:

pgroup The *process group ID* of the application (see "Process Groups" on page 4-27 for more information), or 0 to specify the application of the calling process. If the specified process group ID is not the process group ID of the calling process, the calling process's user ID must either be *root* or the same user ID as the specified application.

priority The new priority, an integer from 0 to 10 inclusive. 0 is the lowest priority, 10 is the highest.

nx_pri() returns 0, or -1 if any error occurs.

For example, the following Fortran call sets the priority of the calling application to 7:

```
      include 'fnx.h'
      integer n

      n = nx_pri(0, 7)
```

The following C call sets the priority of the application with process group ID 738423 to 0:

```
#include <nx.h>
int n;

n = nx_pri(738423, 0);
```

In each of these examples, the variable *n* is assigned 0, or -1 if any error occurred.

Copying a Process onto the Nodes with nx_nfork()

nx_nfork() copies the process that calls it onto the specified set of nodes with the specified process type. It creates one *child process* on each specified node. **nx_nfork()** is similar to the standard OSF/1 call **fork()** except that it can fork processes onto multiple nodes and specifies the process type for the child processes. The parameters of **nx_nfork()** have the following meanings:

node_list An array of integers, each of which specifies a node number within the application (no node number may appear more than once in this array). The calling process is copied onto each of the specified nodes.

numnodes The number of node numbers in *node_list*, or -1 to use all the nodes in the application (in which case *node_list* is ignored).

ptype The process type for each child process.

pid_list An array of integers, into which are stored the OSF/1 process identifiers (PIDs) of the child processes. This array is only for the parent process. The child process get a zero-filled array. See "Using PIDs" on page 4-20 for more information.

nx_nfork() returns the number of child processes created to the parent process and 0 to each child process, or -1 if any error occurs.

For example, the following C calls create an application whose characteristics are specified by the user, then copy the calling process onto all nodes of the application. The process type of each child process is set to 0.

```
#include <nx.h>
#include <sys/types.h>

main(int argc, char *argv[]) {
    int n;
    pid_t pids[2000];

    n = nx_initve("", 0, "", &argc, argv);
    n = nx_nfork(NULL, -1, 0, pids);
```

Note that the *node_list* argument is ignored when the *numnodes* argument is -1, so you can specify a **NULL** pointer in this case (in Fortran, you can use the value 0). After the call to **nx_nfork()**, the variable *n* contains the number of child processes created, or -1 if any error occurred; the first *n* elements of the array *pids* contains the PIDs of the child processes. If more than 2000 child processes are created, unexpected results will occur.

The following Fortran calls create an application with 100 nodes and copy the calling process onto the first 50 nodes of the application (nodes 0 through 49). The process type of each child process is set to 0.

```
      include 'fnx.h'
      integer n
      integer nodes(50), pids(50)

      n = nx_initve("mypart", 100, "", 0, 0)

      do 2, i = 1, 50
         nodes(i) = i - 1
2     continue

      n = nx_nfork(nodes, 50, 0, pids)
```

After the call to **nx_nfork()**, the variable *n* contains 50, or -1 if any error occurred; the array *pids* contains the PIDs of the child processes.

Loading a Program onto the Nodes with nx_load()

nx_load() executes the specified file on the specified set of nodes with the specified process type. Like **nx_nfork**(), **nx_load**() creates one child process on each specified node. The parameters of **nx_load**() have the following meanings:

node_list An array of integers, each of which specifies a node number within the application (no node number may appear more than once in this array). The specified file is loaded onto each of the specified nodes.

numnodes The number of node numbers in *node_list*, or -1 to use all the nodes in the application (in which case *node_list* is ignored).

ptype The process type for each child process.

pid_list An array of integers, into which are stored the OSF/1 process identifiers (PIDs) of the child processes. See "Using PIDs" on page 4-20 for more information.

pathname The relative or absolute pathname of the file to load.

nx_load() returns the number of child processes created, or -1 if any error occurs.

For example, the following Fortran calls create an application whose characteristics are specified by the user, then load and start the program *myprog* on all nodes of the application. The process type of each child process is set to 0.

```
include 'fnx.h'
integer n
integer pids(2000)

n = nx_initve("", 0, "", 1, 0)
n = nx_load(0, -1, 0, pids, "myprog")
```

After the call to **nx_load**(), the variable *n* contains the number of child processes created, or -1 if any error occurred; the first *n* elements of the array *pids* contains the PIDs of the child processes. If more than 2000 child processes are created, unexpected results will occur.

The following C calls create an application with 10 nodes in the partition *mypart*, then load and start the program *../bin/myprog* on nodes 1, 5, and 7 of the application. The process type of each child process is set to 1.

```
#include <nx.h>
#include <sys/types.h>
int   n, i;
int   nodes[3];
pid_t pids[3];

i = 0;
n = nx_initve("mypart", 10, "", &i, NULL);

nodes[0] = 1;
nodes[1] = 5;
nodes[2] = 7;

n = nx_load(nodes, 3, 1, pids, "../bin/myprog");
```

After the call to **nx_load()**, the variable *n* contains 3, or -1 if any error occurred; the array *pids* contains the PIDs of the child processes.

Loading a Program onto the Nodes with nx_loadve()

nx_loadve() is just like **nx_load()** except that it also lets you specify the argument list and environment variables for the new processes (in C). **nx_loadve()** has the following additional parameters:

argv In C, this parameter contains the command line for the child process (you can use the *argv* parameter of **main()** or construct the command line yourself).

envp In C, this parameter contains the environment variables for the child process (you can use the *envp* parameter of **main()** or construct the environment yourself).

In Fortran, you must specify the value 0 for the *argv* and *envp* parameters (or use **nx_load()** instead). This is necessary because these parameters are pointers to arrays of strings, which have no equivalent in Fortran.

nx_loadve() returns the number of child processes created, or -1 if any error occurs. If an error occurs, the value 0 is also stored into the *pid_list* for each process that was not successfully started.

For example, the following C calls create an application as specified by the user (if not specified, the default number of nodes in the default partition), then set the value of the environment variable *HOME* to */tmp*, then load and start the program *myprog* on all nodes of the application with process type 0:

```
#include <nx.h>
#include <stdlib.h>
#include <sys/types.h>
extern char **environ;

main(int argc, char *argv[]) {
    int n;
    pid_t pids[2000];

    n = nx_initve(NULL, 0, NULL, &argc, argv);
    putenv("HOME=/tmp");
    n = nx_loadve(NULL, -1, 0, pids, "myprog", argv, environ);
```

The argument list of *myprog* consists of any command-line arguments to the calling program that were not recognized and removed by **nx_initve()**, and the environment of *myprog* is the same as the user's environment except for the value of *HOME*.

Waiting for Application Processes with nx_waitall()

nx_nfork(), **nx_load()**, and **nx_loadve()** return immediately to the calling process. To wait for the processes created by **nx_nfork()**, **nx_load()**, or **nx_loadve()** to complete, call **nx_waitall()**. **nx_waitall()** simply blocks until all the child processes of the calling process have terminated. It returns 0, or -1 if any error occurs.

For example, the following Fortran calls create a new application as specified by the user, run the program *myprog* on all nodes of the application, and wait until all the node processes have completed:

```
      include 'fnx.h'
      integer n
      integer pids(2000)

      n = nx_initve("", 0, "", 1, 0)
      n = nx_load(0, -1, 0, pids, "myprog")
      n = nx_waitall()
```

Using PIDs

The *pid_list* argument of **nx_nfork()**, **nx_load()**, and **nx_loadve()** receives the OSF/1 process identifiers (PIDs) of the child processes created by the call. The specified array must be large enough to hold all the PIDs—that is, it must have at least as many elements as the *maximum* number of processes that could be created by the call. If more child processes are created than the number of elements in the *pid_list*, unexpected results will occur (the program will probably crash).

In the typical case where you create one process per node of the application, you can use the value returned by any of the **nx_initve...()** calls to determine the number of nodes in the application, then use **malloc()** or an equivalent call to dynamically allocate a *pid_list* with the same number of elements. For example, the following example allocates the appropriate number of elements to the array *pids* based on the application size specified by the user in *argv*:

```
#include <nx.h>
#include <stdio.h>
#include <malloc.h>

main(int argc, char **argv) {
    int   nnodes;
    long *pids;

    nnodes = nx_initve(NULL, 0, NULL, &argc, argv);
    pids = (long *)calloc(nnodes, sizeof(long));
    nx_nfork(NULL, -1, 0, pids);
```

If you don't use dynamic allocation, you should give the *pid_list* as many elements as the number of nodes on the largest system on which the application will be run. For portability to very large Paragon supercomputers, this array should have at least 1000 elements (and possibly more in the future).

Each element in the *pid_list* receives the PID of the process on the node specified by the corresponding element of the *node_list*. If *numnodes* is -1, the PID of the process on node 0 is stored into the first element of *pid_list*, the PID of the process on node 1 is stored into the second element of *pid_list*, and so on. If one or more processes were not successfully started, the value 0 is stored into the corresponding element of the *pid_list*.

NOTE

The PIDs stored into the *pid_list* are OSF/1 PIDs, not process types.

OSF/1 PIDs are unique throughout the system; they are used with standard OSF/1 system calls such as **kill()**. (Note that **kill()** and other system interface routines are supported by the Fortran runtime library; see the *Paragon*™ *System Fortran Compiler User's Guide* for information on these routines.) Process types are unique only within a single application and a single node; they are used with message-passing calls such as **csend()**.

For example, the following C calls create an application as specified by the user, run the program *myprog* on all nodes of the application with process type 0, and then send the signal **SIGKILL** to all the node processes:

```
#include <nx.h>
#include <signal.h>
#include <sys/types.h>

main(int argc, char *argv[]) {
    int n, i;
    pid_t pids[2000];

    n = nx_initve(NULL, 0, NULL, &argc, argv);
    n = nx_load(NULL, -1, 0, pids, "myprog");

    for(i=0; i<n; i++) {
        kill(pids[i], SIGKILL);
    }
```

Getting Information About Applications

Synopsis	**Description**
nx_app_rect(*rows*, *cols***)**	Obtain the height and width of the rectangle of nodes allocated to the current application.
nx_app_nodes(*pgroup*, *node_list*, *list_size***)**	List the nodes allocated to an application.
nx_pspart(*partition*, *pspart_list*, *list_size***)**	Obtain information about all applications and active subpartitions in a partition (C only).

To get information about applications once they are running, use **nx_app_rect()**, **nx_app_nodes()**, and **nx_pspart()**. **nx_app_rect()** returns the application's shape (height and width of the rectangle of nodes allocated to the application); **nx_app_nodes()** returns a list of the nodes that are allocated to the application; and **nx_pspart()** returns information about all the active applications and subpartitions in a partition (like the **pspart** command).

NOTE

Do not call **nx_app_nodes()** or **nx_pspart()** on more than a few nodes at once.

If many nodes use the application information calls at the same time, the allocator daemon can become overwhelmed with requests, which could slow down your application or reduce system stability. If all the nodes in your application need this information, you should have one node make the call and then distribute the information to the other nodes. Note, though, that **nx_app_rect()** is not subject to this restriction.

Finding an Application's Shape with nx_app_rect()

Sometimes, in addition to its node number and the number of nodes in the application, a process needs to know the *shape* of the application. For example, an application might use a different message-passing algorithm depending on whether the nodes allocated to the application form a square, a tall skinny rectangle, a short wide rectangle, or something else (such as a group of noncontiguous nodes).

To find out the rectangular dimensions of the nodes allocated to the application, call **nx_app_rect()**. **nx_app_rect()** stores the height of the application into *rows* and the width of the application into *cols*. If the nodes allocated to the application do not form a rectangle, **nx_app_rect()** stores 1 into *rows* and **numnodes()** into *cols*. **nx_app_rect()** returns 0 if it is successful, or -1 if any error occurs.

For example, the following code fragment in Fortran stores the height of the application into *y* and the width of the application into *x*:

```
integer*4 x, y, result
    •
    •
    •
result = nx_app_rect(y, x)
```

The following code fragment in C does the same:

```
long x, y, result;
    •
    •
    •
result = nx_app_rect(&y, &x);
```

See "Specifying a Rectangle of Nodes" on page 2-16 for information on how to run your application on a rectangular group of nodes with a specific shape.

NOTE

nx_app_rect() can also be called by the name **mypart()** for compatibility with the Touchstone DELTA System.

Listing an Application's Nodes with nx_app_nodes()

Occasionally you want to know an application's physical location within the system. You can use this information to help track down possible hardware problems or make use of nodes with special hardware features (such as extra memory or special I/O interfaces).

To list the nodes allocated to an application, call **nx_app_nodes()**. **nx_app_nodes()** has the following parameters:

pgroup The process group ID of the application (see "Process Groups" on page 4-27 for more information), or 0 to specify the application of the calling process. If the specified process group ID is not the process group ID of the calling process, the calling process's user ID must either be *root* or the same user ID as the specified application.

node_list Pointer variable into which **nx_app_nodes()** stores the address of the list of nodes. The call allocates the memory for this list; when you are finished using the information, you should release this memory by calling **free()**.

list_size Variable into which **nx_app_nodes()** stores the number of entries in *node_list*.

The node numbers returned by **nx_app_nodes()** are node numbers from the root partition, which tell you where in the machine the application is located. **nx_app_nodes()** returns 0 for success, or -1 if any error occurs.

For example, the following Fortran program fragment prints the root node numbers of the nodes on which the current application is running:

```
include 'fnx.h'

integer*4  mynodes(1)
pointer    (ptr, mynodes)
integer    nnodes
integer    i, status

status = nx_app_nodes(0, ptr, nnodes)

if(status .ne. 0) then
   call nx_perror("nx_app_nodes()")
   stop
```

```
      end if

      do 2, i = 1, nnodes
         print *, mynodes(i)
2     continue

      call free(ptr)
```

The equivalent C code is as follows:

```
#include <nx.h>

nx_nodes_t     mynodes;
unsigned long  nnodes;
int            i, status;

status = nx_app_nodes(0, &mynodes, &nnodes);

if(status != 0) {
    nx_perror("nx_app_nodes()");
    exit(1);
}

for(i = 0; i < nnodes; i++) {
    printf("%d\n", mynodes[i]);
}

free(mynodes);
```

Note the use of the & operator on the variables *mynodes* and *nnodes* in the call to **nx_app_nodes()**.

Listing the Applications in a Partition with nx_pspart()

nx_pspart() returns information about each of the applications and subpartitions in a partition, like the **pspart** command. It is callable only from C, not Fortran. It has the following parameters:

partition The relative or absolute pathname of the partition. The specified partition must exist and must give read permission to the calling process.

pspart_list Pointer variable into which **nx_pspart()** stores the address of an array of *nx_pspart_t* structures. Each structure in the array describes one object (application or subpartition). The *nx_pspart_t* structure is defined in *allocsys.h*, which is automatically included by *nx.h* and *fnx.h*. It includes the following fields:

object_type The type of the object described by this structure: **NX_APPLICATION** or **NX_PARTITION**. (These are constants defined in *nx.h* or *fnx.h*).

object_id If the object is an application, this is its process group ID. If the object is a partition, this is an arbitrary value and should be ignored.

uid The numeric user ID of the object's owner.

gid The numeric group ID of the object's group.

size The number of nodes allocated to the object.

priority The current priority of the object.

rolled_in The amount of time the object has been rolled in during the current rollin quantum, expressed as an integer number of milliseconds.

rollin_q The rollin quantum for the object's parent partition (that is, the partition specified in the **nx_pspart()** call), expressed as an integer number of milliseconds.

elapsed The total amount of time the object has been rolled in since it was started, expressed as an integer number of milliseconds.

active Whether or not the object is currently active (rolled-in), inactive (rolled-out), or is dumping core: 0 if the object is inactive, 1 if the object is active, 2 if the object is inactive and is either dumping core or has dumped core, or 3 if the object is active and is either dumping core or has dumped core.

time_started The time the object was started, as returned by **time()**. (If the object is a subpartition, the time the oldest application in the subpartition was started.)

nx_pspart() allocates the memory for the *pspart_list* array; when you are finished using the information, you should release this memory by calling **free()**.

list_size Variable into which **nx_pspart()** stores the number of *nx_pspart_t* structures in *pspart_list.*

nx_pspart() returns 0 for success, or -1 if any error occurs.

For example, the following program fragment prints the numeric user ID and size for every application and subpartition in the partition *mypart*:

```
#include <nx.h>

nx_pspart_t    *info;
unsigned long  nobjs;
int            status, i;

status = nx_pspart("mypart", &info, &nobjs);

if(status != 0) {
    nx_perror("nx_pspart()");
    exit(1);
}

for(i = 0; i < nobjs; i++) {
    printf("uid = %d, size = %d\n", info->uid, info->size);
}

free(info);
```

Note the use of the & operator on the structure *info* and the variable *nobjs* in the call to **nx_pspart()**.

The Controlling Process

By calling any of the **nx_initve...()** calls, a process creates a new application. The process that called **nx_initve...()** becomes the new application's *controlling process*. Each application has exactly one controlling process, and each process controls at most one application.

The controlling process is a special process that creates and controls the application:

- The controlling process can create new processes in the application, using the function **nx_nfork()**, **nx_load()**, or **nx_loadve()**.

- The controlling process can wait for an application process to complete, using **nx_waitall()** or the standard OSF/1 function **wait()** or **waitpid()**.

- The controlling process can send a signal to an application process, or terminate it, using the standard OSF/1 function **kill()**. In particular, the controlling process can send a signal to all the processes in the application (including itself) by using **kill(0,** *signal*).

You can terminate the entire application by terminating the controlling process, using the **kill** command or your interrupt key (normally `<Ctrl-c>` or `<Del>`). The controlling process always runs in the service partition; the application processes run in the partition specified by **nx_initve...()**. If the application processes are running in a gang-scheduled partition, the controlling process is rolled in and out along with its application (that is, when the application is rolled out, the controlling process gets no processor time; when the application is rolled in, the controlling process gets its normal share of the service partition's processor time).

NOTE

Interrupting or suspending an application that is "rolled out" will not take effect until the application is "rolled in" again.

In OSF/1 terms, the controlling process is a *parent* process and the processes created by **nx_nfork()**, **nx_load()**, or **nx_loadve()** are its *child* processes. (In this respect, **nx_nfork()** is similar to **fork()**, **nx_load()** is similar to a **fork()** followed by an **execv()** with a null argument list, and **nx_loadve()** is similar to a **fork()** followed by an **execve()**). The controlling process and the application processes all belong to the same *process group*, and the controlling process is the *process group leader* of the group. No process outside the application belongs to this process group.

The controlling process does not usually do heavy computational work, because it runs in the service partition along with users' shells and other interactive processes. Since it is scheduled interactively, the controlling process will not give as much throughput as application processes running in gang-scheduled compute partitions.

See the *OSF/1 Programmer's Reference* for information on **wait()**, **waitpid()**, **kill()**, **fork()**, and **execve()**.

Process Groups

Process groups are a standard OSF/1 concept. A process group is a set of related processes. You can send a signal to all the processes in a group at once with **kill()**, and you can wait for any process in a group with **waitpid()**. The processes in a process group also share access to a terminal, called the *controlling terminal* of the group. Each process belongs to exactly one process group.

The processes in a process group are all children (or grandchildren, and so on) of the oldest process in the group, called the *process group leader*. The process group leader's process ID (PID) is used to identify the group, and is also called the *process group ID* of the whole group. (Note that this is the process group leader's OSF/1 PID, not its process type.) A process can determine its process group ID by calling **getpgrp()**.

Normally, a process belongs to the same process group as its parent process. However, a process can leave its parent's process group and start a new process group of its own by making such calls as **setpgid()**, **setpgrp()**, or **setsid()**. These calls create a new process group, then remove the calling process from its current group and place it in the new group. The calling process becomes the new group's process group leader, and the caller's PID becomes the new group's process group ID. After that, any processes created by the process group leader belong to the new process group. See the *OSF/1 Programmer's Reference* for information on **setpgid()** and **getpgrp()**.

Process Groups in the Operating System

In the operating system on a Paragon system, process groups work the same as they do in standard OSF/1. In addition, **nx_initve...()** makes sure that the calling process is a process group leader. If the calling process is not already a process group leader, **nx_initve...()** has the same effect as **setpgid()**: it creates a new process group and makes the calling process the new group's process group leader. Because all the processes in the application are created by the controlling process, all the processes in an application are members of the same process group, and no other process in the system is a member of that process group. This means that the application's process group ID uniquely identifies the application, which is why you use a process group ID to identify the application in **nx_pri()**.

This also means that if a process in an application leaves the application's process group by calling **nx_initve...()** (or **setpgid()**, **setpgrp()**, or **setsid()**), it leaves the application. If a process leaves its application's process group, it is no longer considered part of the application and can no longer exchange messages with the other processes in the application. You shouldn't do this unless you know exactly what you are doing.

Killing Application Processes

You can take advantage of the fact that all the processes in the application are members of the same process group by using OSF/1 system calls that affect process groups. For example, specifying a process ID of 0 (zero) to **kill()** sends the specified signal to all the members of the calling process's process group, so the following call kills the entire application (including the calling process):

```
kill(0, SIGKILL);
```

This call differs from the example discussed under "Using PIDs" on page 4-20 in that it also kills the calling process.

An Example Controlling Process

The following C program (which must be linked with **-lnx**, not **-nx**) copies itself onto eight nodes of the partition *mypart* with a process type of 0 and a priority of 7. The eight application processes print "Hello from node *n*" and then exit. The controlling process waits for the application processes to finish, then prints "Hello from controlling process" before exiting itself. Note that this program is executed by both the controlling process and the application processes.

```
#include <nx.h>
#include <sys/types.h>
#include <stdio.h>
#define NUMNODES 8

main(int argc, char **argv) {
    int   n, i;
    pid_t pids[NUMNODES];

    /* create application */
    n = nx_initve("mypart", NUMNODES, NULL, &argc, argv);
    if(n == -1) {
        /* nx_initve() failed */
        perror("nx_initve");
        exit(1);
    }

    /* set application priority to 7 */
    n = nx_pri(0, 7); /* 0 specifies calling application */
    if(n == -1) {
        /* nx_pri() failed */
        perror("nx_pri");
        exit(1);
    }

    /* fork child processes onto all nodes of application */
    n = nx_nfork(NULL, -1, 0, pids);
    if (n == -1) {
        /* nx_nfork() failed */
        perror("nx_nfork");
        exit(1);
    } else if(n == 0) {
        /* child process: print "Hello" and exit */
        printf("Hello from node %d!\n", mynode());
        exit(0);
    } else {
        /* parent (controlling process): wait for all children */
        nx_waitall();
        /* now print "Hello" and exit */
        printf("Hello from controlling process!\n");
        exit(0);
    }
}
```

Message Passing Between Controlling Process and Application Processes

Synopsis	Description
myhost()	Obtain the controlling process's node number.

A controlling process can exchange messages with its child processes using the message-passing system calls described in Chapter 3.

- The controlling process's node number is equal to **numnodes()**. (The maximum node number within the application is **numnodes()** – 1.) The controlling process's node number is also returned by **myhost()** in any process in the application. In the controlling process, **myhost()**, **mynode()**, and **numnodes()** all return the same number.

- The controlling process's process type is initially **INVALID_PTYPE**, but you can change it to a valid value by calling **setptype()**. For best performance, you should not call **setptype()** until *after* you have created all application processes with **nx_nfork()**, **nx_load()**, or **nx_loadve()**, and you should not call **setptype()** at all unless you need to exchange messages with application processes.

Although the controlling process can exchange messages with the application processes, it does not participate in global operations:

- The controlling process may not make any of the calls described under "Global Operations" on page 3-27.

- The controlling process does not participate when the application processes make any of the calls described under "Global Operations" on page 3-27.

- The controlling process does not get messages sent to node number -1 (all nodes).

A send to node -1 (all nodes) sends the message to all the nodes in the application (except the calling process's node), but not the controlling process. This applies whether the message is sent by a node process or by the controlling process itself. On the other hand, an extended receive that specifies node -1 (any node) as the sending node *will* match a message from the controlling process.

Here is an application, written in Fortran, that demonstrates message-passing between the controlling process and the application processes. This application multiplies two numbers (in a very inefficient way). It consists of two programs, *control.f* and *app.f.* You must link *control.f* with **-lnx**, not **-nx**; *app.f* can be linked with either **-lnx** or **-nx**.

The controlling process (*control.f*) requests a number of nodes and an integer value from the user. It creates an application of the specified number of nodes on the partition *mypart* and loads the program *app* onto each node. It then sends the user's integer value to each node as a message (note that the node number -1 sends to all nodes, not including the controlling process) and waits for a return message with the result. When the result is received, the controlling process prints its value and then exits.

```
      include 'fnx.h'

      integer      num_nodes, n, i
      integer      nodes(128), pids(128)
      integer      app_ptype
      parameter    (app_ptype = 0)
      integer      data, result
      integer      result_type, data_type
      parameter    (result_type = 1)
      parameter    (data_type = 2)

c get number of nodes (limited to size of "nodes" and "pids" arrays)
1     print *, "Enter number of nodes (must not be greater than 128)"
      read(*,*) num_nodes
      if(num_nodes .gt. 128) goto 1

c create application of specified size
      n = nx_initve("mypart", num_nodes, "", 0, 0)
      if(n .eq. -1) then
         call nx_perror("nx_initve")
         stop
      end if

c fill in node array for nx_load()
      do 2, i = 1, num_nodes
         nodes(i) = i - 1
2     continue

c load program "app" onto the nodes of the application
      n = nx_load(nodes, num_nodes, app_ptype, pids, "app")
      if(n .eq. -1) then
         call nx_perror("nx_load")
         stop
      end if

c get an integer from the user
      print *, "Enter value to be summed"
      read(*,*) data

c set my process type
      call setptype(app_ptype)

c send integer to all the nodes
      call csend(data_type, data, 4, -1, app_ptype)

c receive the result
      call crecv(result_type, result, 4)
```

```
c print the result
      print *, "The sum of ",data," over ",num_nodes," nodes is ",result

      end
```

The application process (*app.f*) waits for a message and performs a **gisum()** on the value received. (Note that the controlling process does not participate in the **gisum()**.) The process on node 0 sends the result to the controlling process, then all the application processes exit.

```
      include 'fnx.h'

      integer      val, work
      integer      result_type, data_type
      parameter    (result_type = 1)
      parameter    (data_type = 2)

c get an integer from the controlling process
      call crecv(data_type, val, 4)

c sum it over all nodes
      call gisum(val, 1, work)

c if I'm node 0, send the result back to the controlling process
      if(mynode() .eq. 0) call csend(result_type, val, 4, myhost(), 0)

      end
```

Managing Partitions

The operating system provides system calls that let you create and remove partitions, get information about partitions, and change their characteristics, like the **mkpart**, **rmpart**, **showpart**, and **chpart** commands described in Chapter 2. See "Managing Partitions" on page 2-30 for introductory information on partitions.

Making Partitions

Synopsis	Description
nx_mkpart(*partition*, *size*, *type***)**	Create a partition with a particular number of nodes.
nx_mkpart_rect(*partition*, *rows*, *cols*, *type***)**	Create a partition with a particular height and width.
nx_mkpart_map(*partition*, *numnodes*, *node_list*, *type***)**	Create a partition with a specific set of nodes.
nx_mkpart_attr(*partition*, [*attribute*, *value*,]... **NX_ATTR_END)**	Create a partition with specified attributes.

To create a partition, use **nx_mkpart()**, **nx_mkpart_rect()**, **nx_mkpart_map()**, or **nx_mkpart_attr()**. These calls all create a partition, but they use different methods to specify the nodes allocated to the new partition:

- **nx_mkpart()** works like the **mkpart** command's **-sz** *size* switch.
- **nx_mkpart_rect()** works like the **mkpart** command's **-sz** *h***X***w* switch.
- **nx_mkpart_map()** works like the **mkpart** command's **-nd** *nodespec* switch (except that only node numbers can be specified).
- **nx_mkpart_attr()** works like all of the **mkpart** command's switches.

See "Specifying the Nodes Allocated to the Partition" on page 2-46 for more information on the **mkpart** command's **-sz** and **-nd** switches.

These calls have the following parameters:

partition The new partition's relative or absolute pathname. The specified new partition must not exist; the parent partition of the specified new partition must exist and must give write permission to the calling process. See "Partition Pathnames" on page 2-33 for more information on partition pathnames; see "Owner, Group, and Protection Modes" on page 2-38 for more information on partition permissions.

size (**nx_mkpart()** only)
The number of nodes of the new partition, or -1 to specify "all the nodes of the parent partition." If you specify a size smaller than that of the parent partition, the nodes are selected by the system (and are not necessarily contiguous).

nx_mkpart() attempts to allocate a square group of nodes if it can. If this is not possible, it attempts to allocate a rectangular group of nodes that is either twice as wide as it is high or twice as high as it is wide. If this is not possible, it allocates any available nodes. In this case, nodes allocated to the partition may not be contiguous.

rows and *cols* (**nx_mkpart_rect()** only)

The height and width of the new partition. The new partition is a rectangle with the specified number of rows and columns, but its location within the parent partition is selected by the system.

numnodes and *node_list* (**nx_mkpart_map()** only)

The exact node numbers within the parent partition for the new partition. The *node_list* parameter is an array of node numbers; the *numnodes* parameter specifies the number of elements in *node_list*.

type (all except **nx_mkpart_attr()**)

The new partition's scheduling type: **NX_STD** to specify standard scheduling, **NX_GANG** to specify gang scheduling, or **NX_SPS** to specify space sharing. The names **NX_STD**, **NX_GANG**, and **NX_SPS** are defined in *nx.h* and *fnx.h*. See "Scheduling Characteristics" on page 2-39 for more information on the different scheduling types.

[*attribute*, *value*,]... (**nx_mkpart_attr()** only)

The new partition's attributes. Each *attribute* is one of the constants described in Table 4-2; the *value* is the value of the specified attribute. The type of the *value* argument is determined by the value of the preceding *attribute* argument. See "Setting Partition Attributes with nx_mkpart_attr()" on page 4-36 for more information.

NX_ATTR_END (**nx_mkpart_attr()** only)

A constant that marks the end of the list of *attribute*, *value* pairs.

nx_mkpart(), **nx_mkpart_rect()**, **nx_mkpart_map()**, and **nx_mkpart_attr()** return the number of nodes in the new partition, or -1 if any error occurs.

The new partition's owner and group are set to the owner and group of the calling process. Any partition characteristics not specified in the call (such as protection modes and rollin quantum) are set to the same values as the parent partition. Once the partition is created, you can use the **nx_chpart...()** calls to set these characteristics to different values, as discussed under "Changing Partition Characteristics" on page 4-49.

Examples

The following Fortran call creates a new gang-scheduled partition called *newpart* whose parent partition is the compute partition (using a relative partition pathname) and which consists of all the nodes in the compute partition:

```
      include 'fnx.h'
      integer n

      n = nx_mkpart("newpart", -1, NX_GANG)
```

The following C call creates a new space-shared partition called *mypart* whose parent partition is the compute partition (using an absolute partition pathname) and which has 54 nodes:

```
#include <nx.h>
int n;

n = nx_mkpart(".compute.mypart", 54, NX_SPS);
```

The following C call creates a new gang-scheduled partition called *rect* whose parent partition is *mypart* and which is 3 nodes high and 4 nodes wide:

```
#include <nx.h>
int n;

n = nx_mkpart_rect(".compute.mypart.rect", 3, 4, NX_GANG);
```

The following C call creates a new space-shared partition called *corners* whose parent partition is *rect* and which consists of the four nodes at the corners of *rect*:

```
#include <nx.h>
long nodes[4];
int n;

nodes[0] = 0;
nodes[1] = 3;
nodes[2] = 8;
nodes[3] = 11;
n = nx_mkpart_map(".compute.mypart.rect.corners", 4,
                  nodes, NX_SPS);
```

In each of these examples, the variable *n* is assigned the number of nodes in the new partition, or -1 if any error occurred.

Examples of **nx_mkpart_attr()** can be found at the end of the next section.

Setting Partition Attributes with nx_mkpart_attr()

NOTE

If you call **nx_mkpart_attr()** in a Fortran subprogram, you must include *fnx.h* after the subprogram declaration and before the call. This is required for the call to recognize the pre-defined attribute constants (for example, **NX_ATTR_SZ**).

The following table describes the *attribute*s and *value*s you can use with **nx_mkpart_attr()**. The attribute constants, including **NX_ATTR_END**, are defined in <*nx.h*> or <*fnx.h*>.

Table 4-2. Attribute Constants for Use with nx_mkpart_attr() (1 of 3)

Attribute Constant	Type of Following Value (C / Fortran)	Equivalent Switch	Description
NX_ATTR_SZ	long INTEGER	**-sz** *size*	Specifies the size of the new partition, like the *size* argument of **nx_mkpart()**.
NX_ATTR_MAP	long * INTEGER(*)	**-nd**	Specifies the list of nodes for the new partition, like the *node_list* argument of **nx_mkpart_map()**. The *value* is an array of node numbers. Do not specify the same node number more than once in this array. **NX_ATTR_SZ** must also be specified (anywhere in the argument list) to give the length of the array.
NX_ATTR_RECT	long * INTEGER(2)	**-sz** *h***X***w*	Specifies a rectangle for the new partition, like the *rows* and *cols* arguments of **nx_mkpart_rect()**. The *value* is an array of two integers (height first, width second). If **NX_ATTR_ANCHOR** is not specified, the system determines the rectangle's location within the parent partition.

Table 4-2. Attribute Constants for Use with nx_mkpart_attr() (2 of 3)

Attribute Constant	Type of Following Value (C / Fortran)	Equivalent Switch	Description
NX_ATTR_ANCHOR	**long** **INTEGER**	(none)	Specifies the node number of the upper left corner of a rectangle specified with **NX_ATTR_RECT**. If the specified node number is -1, the system determines the rectangle's location within the parent partition. **NX_ATTR_RECT** must also be specified (anywhere in the argument list) to give the height and width of the rectangle.
NX_ATTR_RELAXED	**long** **INTEGER**	**-rlx**	Specifies whether or not the requested number of nodes can be relaxed (see "Relaxing Partition Size" on page 2-49). The value 0 means all requested nodes must be available; the value 1 relaxes this requirement. **NX_ATTR_RELAXED** cannot be used with **NX_ATTR_RECT**, unless **NX_ATTR_ANCHOR** is also specified with a value other than -1.
NX_ATTR_SEL	**char *** **CHARACTER *(*)**	**-nt**	Specifies the node type for the partition. The *value* is a character string whose value is a node type specifier (see "Specifying Node Attributes" on page 2-25). If **NX_ATTR_SZ**, **NX_ATTR_MAP**, or **NX_ATTR_RECT** is also specified, the given nodes must all be available and have the specified node type, unless **NX_ATTR_RELAXED** is specified.
NX_ATTR_SCHED	**long** **INTEGER**	**-ss/-sps/** **-rq/-epl**	Sets the new partition's node type, like the *type* argument of **nx_mkpart()**. The *value* is **NX_STD**, **NX_GANG**, or **NX_SPS** (see "Scheduling Characteristics" on page 2-39 for more information on the different scheduling types).

Table 4-2. Attribute Constants for Use with nx_mkpart_attr() (3 of 3)

Attribute Constant	Type of Following Value (C / Fortran)	Equivalent Switch	Description
NX_ATTR_EPL	**long** **INTEGER**	**-epl** *priority*	Sets the new partition's effective priority limit. The *value* is an integer from 0 to 10 (see "Scheduling Characteristics" on page 2-39). **NX_ATTR_EPL** can be used with or without **NX_MKPART_SCHED**. If **NX_MKPART_SCHED** is used, its value must be set to **NX_GANG**.
NX_ATTR_RQ	**long** **INTEGER**	**-rq** *time*	Sets the new partition's rollin quantum. The *value* is an integer number of milliseconds, or 0 to specify an "infinite" rollin quantum (see "Scheduling Characteristics" on page 2-39). The *value* must be less than or equal to 86,400,000 milliseconds (24 hours) and greater than or equal to the minimum rollin quantum (determined by your system administrator). If it is not a multiple of 100, it is silently rounded up. **NX_ATTR_RQ** can be used with or without **NX_MKPART_SCHED**. If **NX_MKPART_SCHED** is used, its value must be set to **NX_GANG**.

Each *attribute* can appear at most once in the argument list. The order of the *attribute*s in the argument list is not significant.

Specifying the Nodes Allocated to the Partition

nx_mkpart_attr() provides the following ways to specify the nodes allocated to the partition:

- Use **NX_ATTR_SZ** alone.

  Requests the specified number of nodes. If the *value* is 0 or -1, requests all the nodes in the parent partition.

NX_ATTR_SZ attempts to create a square partition if it can. If this is not possible, it attempts to create a rectangular partition that is either twice as wide as it is high or twice as high as it is wide. If this is not possible, it uses any available nodes. In this case, the nodes allocated to the partition may not be contiguous.

- Use both **NX_ATTR_MAP** and **NX_ATTR_SZ**.

  Requests the specified list of nodes. **NX_ATTR_MAP** and **NX_ATTR_SZ** can appear in any order in the argument list.

- Use **NX_ATTR_RECT** alone.

  Requests a rectangular partition of the specified height and width. The system places the rectangle within the parent partition.

- Use both **NX_ATTR_RECT** and **NX_ATTR_ANCHOR**.

  Requests a rectangular partition of the specified height and width, whose upper left corner is located at the specified anchor node within the parent partition. **NX_ATTR_RECT** and **NX_ATTR_ANCHOR** can appear in any order in the argument list. If the value of **NX_ATTR_ANCHOR** is -1, the system places the rectangle within the parent partition.

- Use **NX_ATTR_SEL** alone.

  Requests all the nodes of the specified node type in the parent partition.

- Use **NX_ATTR_SEL** together with **NX_ATTR_SZ**, **NX_ATTR_MAP**, **NX_ATTR_RECT**, and/or **NX_ATTR_ANCHOR**.

  Requests the nodes specified by the **NX_ATTR_SZ**, **NX_ATTR_MAP**, **NX_ATTR_RECT**, and/or **NX_ATTR_ANCHOR**, all of which must have the node type specified by the **NX_ATTR_SEL**. See "Running an Application on a Particular Node Type" on page 2-23 for more information.

- Do not use **NX_ATTR_SEL**, **NX_ATTR_SZ**, **NX_ATTR_MAP**, **NX_ATTR_RECT**, or **NX_ATTR_ANCHOR**.

  Requests all the nodes in the parent partition.

- Use **NX_ATTR_RELAXED** with a *value* of 1 together with any of the above.

  Requests all the *available* nodes in the specified node set, *up to* the number of nodes requested. At least one of the specified nodes must be available. See "Relaxing Application Size" on page 2-17 for more information.

If **NX_ATTR_SEL** is used together with **NX_ATTR_RELAXED**, only nodes of the specified type are returned, but the number of nodes returned may be less than the number of nodes requested. At least one node of the specified type must be available. See "Using Node Attributes with a Relaxed Application Size" on page 2-28 for more information.

The following combinations of these attributes are invalid:

- You cannot use **NX_ATTR_MAP** without **NX_ATTR_SZ**.
- You cannot use **NX_ATTR_ANCHOR** without **NX_ATTR_RECT**.
- You cannot use **NX_ATTR_SZ** or **NX_ATTR_MAP** together with **NX_ATTR_RECT**.
- You cannot use **NX_ATTR_RELAXED** together with **NX_ATTR_RECT**, unless **NX_ATTR_ANCHOR** is also specified with a value other than -1.

Using any of these combinations of attributes causes **nx_mkpart_attr()** to fail with the error "invalid attribute specified."

Examples

The following Fortran call creates a new partition called *newpart* whose parent partition is the compute partition (using a relative partition pathname). The new partition consists of all the nodes in the compute partition and has the same scheduling type, rollin quantum, and effective priority limit as the compute partition:

```
include 'fnx.h'
integer n

n = nx_mkpart_attr("newpart", NX_ATTR_END)
```

The following C call creates a new space-shared partition called *mypart* whose parent partition is the compute partition (using an absolute partition pathname) and which has 54 nodes:

```
#include <nx.h>
int n;

n = nx_mkpart_attr(".compute.mypart",
                   NX_ATTR_SZ, 54,
                   NX_ATTR_SCHED, NX_SPS,
                   NX_ATTR_END);
```

The following C call creates a new gang-scheduled partition called *rect* whose parent partition is *mypart*. It is 3 nodes high and 4 nodes wide, and has its upper left corner at node 1 of *mypart*. It has a rollin quantum of 600,000 milliseconds (10 minutes) and the same effective priority limit as *mypart*:

```
#include <nx.h>
long rect[2];
int n;

rect[0] = 3;
rect[1] = 4;

n = nx_mkpart_attr(".compute.mypart.rect",
                   NX_ATTR_RECT, rect,
                   NX_ATTR_ANCHOR, 1,
                   NX_ATTR_RQ, 600000,
                   NX_ATTR_END);
```

The following C call creates a new gang-scheduled partition called *corners* whose parent partition is *rect* and which consists of the four nodes at the corners of *rect*. It has an effective priority limit of 3. All other characteristics are the same as *rect*:

```
#include <nx.h>
long nodes[4];
int n;

nodes[0] = 0;
nodes[1] = 3;
nodes[2] = 8;
nodes[3] = 11;
n = nx_mkpart_attr(".compute.mypart.rect.corners",
                   NX_ATTR_MAP, nodes,
                   NX_ATTR_SZ, 4,
                   NX_ATTR_EPL, 3,
                   NX_ATTR_END);
```

The following Fortran call creates a new partition called **bigmem** whose parent partition is the compute partition and consists of all *available* nodes with 64M bytes or more of physical RAM. All other characteristics of *bigmem* are the same as those of the compute partition:

```
      include 'fnx.h'
      integer n

      n = nx_mkpart_attr("bigmem",
     x                   NX_ATTR_SEL, ">=64mb",
     x                   NX_ATTR_RELAXED, 1,
     x                   NX_ATTR_END)
```

In each of these examples, the variable *n* is assigned the number of nodes in the new partition, or -1 if any error occurred.

Removing Partitions

Synopsis	Description
nx_rmpart(*partition*, *force*, *recursive*)	Remove a partition.

To remove a partition, use **nx_rmpart**(). The parameters of **nx_rmpart**() have the following meanings:

partition The relative or absolute pathname of the partition to be removed. The parent partition of the specified partition must give write permission to the calling process. See "Partition Pathnames" on page 2-33 for more information on partition pathnames; see "Owner, Group, and Protection Modes" on page 2-38 for more information on partition permissions.

force Specifies whether to remove the partition if it contains running applications: if *force* is 0, the partition will not be removed if it contains any applications; if *force* is any value other than 0, the partition will be removed even if it contains applications.

recursive Specifies whether to remove the partition if it contains subpartitions: if *recursive* is 0, the partition will not be removed if it contains any subpartitions; if *recursive* is any value other than 0, the partition will be removed along with all its subpartitions, sub-subpartitions, and so on. This is an "all or nothing" operation: if any subpartitions cannot be removed, the call fails and no subpartitions are removed.

If the partition contains both subpartitions and applications, or contains subpartitions that contain applications, you must set both *force* and *recursive* to a nonzero value to remove it.

nx_rmpart() returns 0 for success, or -1 if any error occurs.

For example, the following Fortran call removes the partition called *newpart* whose parent partition is the compute partition (using a relative partition pathname), but only if it does not contain any running applications or subpartitions:

```
include 'fnx.h'
integer n

n = nx_rmpart("newpart", 0, 0)
```

After this call, the variable *n* contains 0 if the partition was removed, or -1 if it was not removed for any reason (for example, if the partition contained applications or subpartitions).

The following C call removes the partition called *mypart* whose parent partition is the compute partition (using an absolute partition pathname), even if it contains running applications; however, it does not remove *mypart* if the partition contains subpartitions:

```
#include <nx.h>
int n;

n = nx_rmpart(".compute.mypart", 1, 0);
```

After this call, the variable *n* contains 0 if the partition was removed, or -1 if it was not removed for any reason (for example, if the partition contained subpartitions, or if the partition does not exist).

Getting Information About Partitions

Synopsis	Description
nx_part_attr(*partition*, *attributes***)**	Get a partition's attributes.
nx_part_nodes(*partition*, *node_list*, *list_size***)**	List the root node numbers for the nodes of a partition.
nx_node_attr(*partition*, *attributes***)**	Get the node attributes for all nodes in a partition (C only).

To get information about a partition, use **nx_part_attr()**, **nx_part_nodes()**, or **nx_node_attr()**. **nx_part_attr()** returns the attributes of a partition, **nx_part_nodes()** returns a list of the nodes in a partition, and **nx_node_attr()** returns the node attributes of the nodes in a partition.

NOTE

Do not call **nx_part_attr()** or **nx_part_nodes()** on more than a few nodes at once.

If many nodes use the partition information calls at the same time, the allocator daemon can become overwhelmed with requests, which could slow down your application or reduce system stability. If all the nodes in your application need this information, you should have one node make the call and then distribute the information to the other nodes.

Determining a Partition's Attributes with nx_part_attr()

nx_part_attr() returns the attributes of a partition. It has the following parameters:

partition The relative or absolute pathname of the partition. The specified partition must exist and must give read permission to the calling process.

attributes A structure of type *nx_part_info_t* (you must allocate the space for this structure). The *nx_part_info_t* structure is defined in *allocsys.h*, which is automatically included by *nx.h* and *fnx.h*. It includes the following elements:

uid The numeric user ID of the partition's owner.

gid The numeric group ID of the partition's group.

access The access permissions of the partition, expressed as a three-digit octal number.

sched The scheduling type of the partition: **NX_STD**, **NX_GANG**, or **NX_SPS**. (These are constants defined in *nx.h* or *fnx.h*).

rq The rollin quantum of the partition, expressed as an integer number of milliseconds (0 for a standard-scheduled or space-shared partition).

epl The effective priority limit of the partition (20 for a standard-scheduled partition).

nodes The number of nodes in the partition.

mesh_x The width of the partition (columns), or -1 if the partition is not rectangular.

mesh_y The height of the partition (rows), or -1 if the partition is not rectangular.

enclose_mesh_x The width of the smallest rectangle that completely encloses the partition.

enclose_mesh_y The height of the smallest rectangle that completely encloses the partition.

nx_part_attr() returns 0 for success, or -1 if any error occurs.

For example, the following C program fragment prints the rollin quantum and effective priority limit for the partition *mypart*:

```
#include <nx.h>

nx_part_info_t info;
int            status;
```

```
    status = nx_part_attr("mypart", &info);

    if(status != 0) {
        nx_perror("nx_part_attr()");
        exit(1);
    }

    printf("rq = %d, epl = %d\n", info.rq, info.epl);
```

Note the use of the & operator on the structure *info* in the call to **nx_part_attr()**. The equivalent Fortran code is as follows:

```
          include 'fnx.h'

          record /nx_part_info_t/ info
          integer status

          status = nx_part_attr("mypart", info)

          if(status .ne. 0) then
             call nx_perror("nx_part_attr()")
             stop
          end if

          print *, "rq =",info.rq,", epl =",info.epl
```

If the partition is not a contiguous rectangle, the values of *mesh_x* and *mesh_y* are -1 and the rectangle described by *enclose_mesh_x* and *enclose_mesh_y* includes nodes that are not part of the partition. For example, Figure 4-1 shows a non-rectangular partition called *mypart.* For this partition:

- *nodes* is 4.
- *mesh_x* and *mesh_y* are both -1.
- *enclose_mesh_x* is 3.
- *enclose_mesh_y* is 2.

Determining a Partition's Nodes with nx_part_nodes()

nx_part_nodes() returns a list of the nodes in the specified partition. You might want to do this to determine whether or not the partition includes a certain node which has special hardware characteristics such as extra memory or an I/O interface. **nx_part_nodes()** has the following parameters:

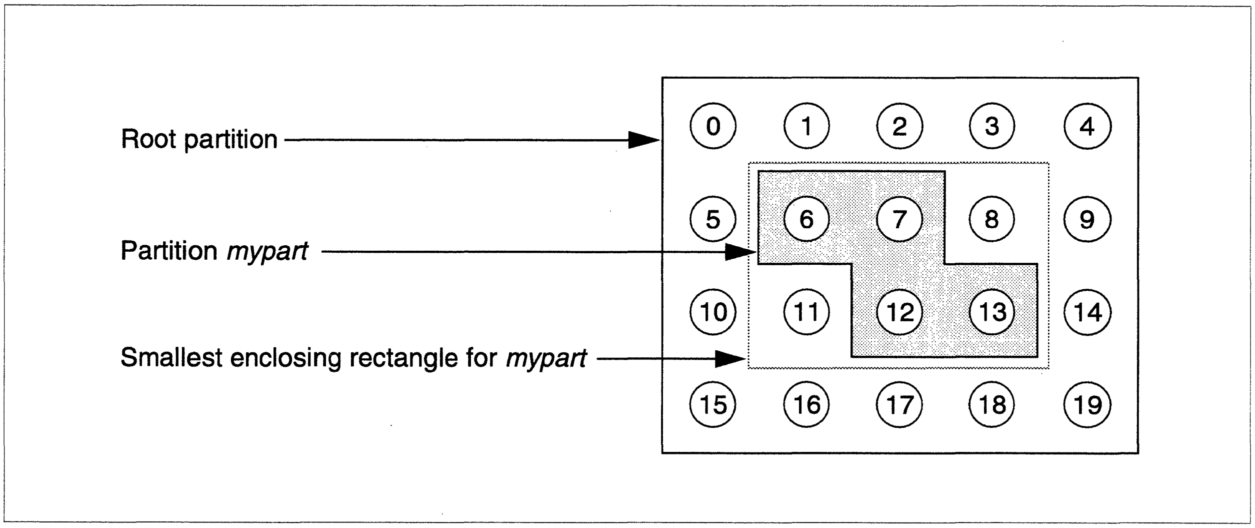


Figure 4-1. Sample Partition for nx_part_attr() and nx_part_nodes()

partition The relative or absolute pathname of the partition. The specified partition must exist and must give read permission to the calling process.

node_list Pointer variable into which **nx_part_nodes()** stores the address of the list of nodes. **nx_part_nodes()** allocates the memory for this list; when you are finished using the information, you should release this memory by calling **free()**.

list_size Variable into which **nx_part_nodes()** stores the number of entries in *node_list*.

nx_part_nodes() returns 0 for success, or -1 if any error occurs.

The node numbers returned by **nx_part_nodes()** are node numbers from the root partition. For example, **nx_part_nodes()** for the partition *mypart* shown in Figure 4-1 would return node numbers 6, 7, 12, and 13. This is true even if the root partition is not the direct parent partition of *mypart*.

For example, the following Fortran program fragment prints the root node numbers for the partition *mypart*:

```
include 'fnx.h'

integer*4  mynodes(1)
pointer    (ptr, mynodes)
integer    nnodes
integer    i, status

status = nx_part_nodes("mypart", ptr, nnodes)
```

```
      if(status .ne. 0) then
         call nx_perror("nx_part_nodes()")
         stop
      end if

      do 2, i = 1, nnodes
         print *, mynodes(i)
2     continue

      call free(ptr)
```

The equivalent C code is as follows:

```
#include <nx.h>

nx_nodes_t     mynodes;
unsigned long  nnodes;
int            i, status;

status = nx_part_nodes("mypart", &mynodes, &nnodes);

if(status != 0) {
    nx_perror("nx_part_nodes()");
    exit(1);
}

for(i = 0; i < nnodes; i++) {
    printf("%d\n", mynodes[i]);
}

free(mynodes);
```

Note the use of the & operator on the variables *mynodes* and *nnodes* in the call to **nx_part_nodes()**.

Determining Node Attributes with nx_node_attr()

nx_node_attr() returns the node attribute strings for every node in a partition. It is callable only from C, not Fortran. It has the following parameters:

partition The relative or absolute pathname of the partition. The specified partition must exist and must give read permission to the calling process.

attributes Address of a variable of type **char ****, into which **nx_node_attr()** stores the address of an array of strings. Each string in this array is a comma-separated list of node attributes for a single node. The call allocates the memory for this array; when you are finished using the information, you should release this memory by calling **free()**.

If successful, **nx_node_attr()** returns the number of nodes for which information is returned. If any error occurs, **nx_node_attr()** returns -1 and sets *errno* to indicate the cause of the error.

NOTE

Do not call **nx_node_attr()** on more than a few nodes at once.

The offset of each string in the returned array corresponds to the node number within the partition; for example, if the variable pointed to by the *attributes* parameter is called *x*, then *x[5]* describes node 5 of the specified partition. Each string is a comma-separated list of *node attributes* (strings that describe the physical properties of the node); see Table 2-1 on page 2-24 for a list of the most commonly-seen node attributes. The node attributes are listed in an arbitrary order within each string.

For example, the following C program fragment prints the node attributes for each node in the partition *mypart*:

```
#include <nx.h>

char  *partition = "mypart";
char **node_attrs;
int    nnodes, i;

nnodes = nx_node_attr(partition, &node_attrs);

if(nnodes == -1) {
    nx_perror("nx_node_attr() failed");
    exit(1);
}

printf("Partition \"%s\" has %d nodes:\n", partition, nnodes);

for(i = 0; i < nnodes; i++) {
    printf("    Node %d: \"%s\"\n", i, node_attrs[i]);
}

free(node_attrs);
```

Note the use of the `&` operator on the variable *node_attrs* in the call to **nx_node_attr()**, and the use of **free()** to free the memory after it is used. The output of this code might be something like this:

```
Partition "mypart" has 7 nodes:
    Node 0: "2proc,64mb,MP"
    Node 1: "2proc,64mb,MP"
    Node 2: "2proc,64mb,MP"
    Node 3: "2proc,64mb,MP"
    Node 4: "2proc,64mb,MP"
    Node 5: "2proc,64mb,MP"
    Node 6: "2proc,64mb,MP"
```

Changing Partition Characteristics

Synopsis	Description
nx_chpart_name(*partition*, *name***)**	Change a partition's name.
nx_chpart_mod(*partition*, *mode***)**	Change a partition's protection modes.
nx_chpart_epl(*partition*, *priority***)**	Change a partition's effective priority limit.
nx_chpart_rq(*partition*, *rollin_quantum***)**	Change a partition's rollin quantum.
nx_chpart_owner(*partition*, *owner*, *group***)**	Change a partition's owner and group.
nx_chpart_sched(*partition*, *sched_type***)**	Change a partition's scheduling type.

To change a partition's characteristics, use **nx_chpart_name()**, **nx_chpart_mod()**, **nx_chpart_epl()**, **nx_chpart_rq()**, **nx_chpart_owner()**, or **nx_chpart_sched()**. Each of these calls changes one characteristic, and leaves the other characteristics unchanged. These calls have the following parameters:

partition The relative or absolute pathname of the partition to change. The specified partition must exist; the permissions required depend on the operation.

name (**nx_chpart_name()** only)

The new name for the partition, expressed as a string of any length containing only uppercase letters, lowercase letters, digits, and underscores. Note that **nx_chpart_name()** can only change the partition's name "in place;" there is no way to move a partition to a different parent partition.

The calling process must have write permission on the parent partition of the specified partition to use **nx_chpart_name()**.

mode (**nx_chpart_mod()** only)

The new protection modes of the partition, expressed as an octal number. See **chmod()** in the *OSF/1 Programmer's Reference* for more information on specifying protection modes; see "Owner, Group, and Protection Modes" on page 2-38 for more information on protection modes for partitions.

The calling process must be the owner of the partition or the system administrator to use **nx_chpart_mod()**.

priority (**nx_chpart_epl()** only)

The new effective priority limit for the partition, expressed as an integer from 0 to 10 inclusive. See "Scheduling Characteristics" on page 2-39 for more information on effective priority limits.

The calling process must have write permission for the partition to use **nx_chpart_epl()**.

rollin_quantum (**nx_chpart_rq()** only)

The new rollin quantum for the partition, expressed as an integer number of milliseconds, or 0 to specify an "infinite" rollin quantum. The specified value must not be greater than 86,400,000 milliseconds (24 hours) and must not be less than the minimum rollin quantum for your system (determined by your system administrator). If it is not a multiple of 100, it is silently rounded up to the next multiple of 100. See "Scheduling Characteristics" on page 2-39 for more information on rollin quanta.

The calling process must have write permission for the partition to use **nx_chpart_rq()**.

owner and *group* (**nx_chpart_owner()** only)

The new user and group for the partition, expressed as a numeric user ID (UID) and group ID (GID). You can also specify -1, meaning "leave owner/group unchanged," for either or both. See "Owner, Group, and Protection Modes" on page 2-38 for more information on partition ownership.

The permissions required for **nx_chpart_owner()** depend on the operation. To change the partition's ownership, the calling process must be the system administrator. To change the partition's group, the calling process must either be the system administrator or must be the partition's owner and changing the group to a group that the calling process belongs to.

sched_type (**nx_chpart_sched()** only)

The new scheduling type for the partition, which must be **NX_GANG** or **NX_SPS** (constants defined in *nx.h* or *fnx.h*). See "Scheduling Characteristics" on page 2-39 for more information on gang-scheduling and space sharing.

The specified partition must not be standard-scheduled. A space-shared partition can be changed to gang-scheduled at any time; a gang-scheduled partition can only be changed to space-shared if it contains no applications and no overlapping subpartitions.

The calling process must have write permission for the partition to use **nx_chpart_sched()**.

nx_chpart_name(), **nx_chpart_mod()**, **nx_chpart_epl()**, **nx_chpart_rq()**, **nx_chpart_owner()**, and **nx_chpart_sched()** return 0 for success, or -1 if any error occurs.

For example, the following Fortran call changes the name of *mypart* to *newpart*:

```
      include 'fnx.h'
      integer n

      n = nx_chpart_name("mypart", "newpart")
```

The following C call has the same effect, but uses an absolute partition pathname:

```
#include <nx.h>
int n;

n = nx_chpart_name(".compute.mypart", "newpart");
```

Note that the second parameter of **nx_chpart_name()** is always a partition name, never a partition pathname. There is no way to move a partition from one parent partition to another.

The following C call sets the permissions of *mypart* to `rwxr-x---` (750 octal):

```
#include <nx.h>
int n;

n = nx_chpart_mod("mypart", 0750);
```

The following Fortran call has the same effect, but uses an absolute partition pathname:

```
      include 'fnx.h'
      integer n
      n = nx_chpart_mod(".compute.mypart", '750'O)
```

The following C call sets *mypart*'s effective priority limit to 7:

```
#include <nx.h>
int n;

n = nx_chpart_epl("mypart", 7);
```

The following Fortran call sets *mypart*'s rollin quantum to 10 minutes (600,000 microseconds):

```
      include 'fnx.h'
      integer n

      n = nx_chpart_rq("mypart", 600000)
```

The following C calls set *mypart*'s owner to *fred* and its group to *devel* (see the *OSF/1 Programmer's Reference* for information on **getpwnam()** and **getgrnam()**, which get the numeric user and group IDs based on their names):

```
#include <stdio.h>
#include <pwd.h>
#include <grp.h>
#include <nx.h>

struct passwd *user;
struct group *group;
int n;

user = getpwnam("fred");
group = getgrnam("devel");
n = nx_chpart_owner("mypart", user->pw_uid, group->gr_gid);
```

The following Fortran call changes *mypart* to a gang-scheduled partition (it must currently be either gang-scheduled or space-shared):

```
      include 'fnx.h'
      integer n

      n = nx_chpart_sched("mypart", NX_GANG)
```

In each of these examples, the variable *n* is assigned 0 if the call succeeded, or -1 if any error occurred.

Listing Unusable Nodes

Synopsis	Description
nx_empty_nodes(*node_list*, *list_size***)**	List the nodes that are empty slots.
nx_failed_nodes(*node_list*, *list_size***)**	List the nodes that failed to boot.

To find out which nodes in the system are unusable, use **nx_empty_nodes()** and **nx_failed_nodes()**. (See "Unusable Nodes" on page 2-37 for more information on unusable nodes.)

- **nx_empty_nodes()** returns a list of the nodes that are part of the root partition but do not have a node board installed in the corresponding slot (these are shown as "-" in the output of **showpart**).
- **nx_failed_nodes()** returns a list of the nodes that are part of the root partition but failed to boot for some reason (these are shown as "X" in the output of **showpart**).

NOTE

Do not call **nx_empty_nodes()** or **nx_failed_nodes()** on more than a few nodes at once.

If many nodes use these calls at the same time, the allocator daemon can become overwhelmed with requests, which could slow down your application or reduce system stability. If all the nodes in your application need this information, you should have one node make the call and then distribute the information to the other nodes.

Both these calls have the following parameters:

node_list Pointer variable into which the call stores the address of the list of nodes. The call allocates the memory for this list; when you are finished using the information, you should release this memory by calling **free()**.

list_size Variable into which the call stores the number of entries in *node_list.*

The node numbers returned by these calls are node numbers from the root partition. Both calls return 0 for success, or -1 if any error occurs.

For example, the following Fortran program fragment prints the node numbers of all empty slots in the root partition:

```
      include 'fnx.h'

      integer*4  empty(1)
      pointer    (ptr, empty)
      integer    nempty
      integer    i, status

      status = nx_empty_nodes(ptr, nempty)

      if(status .ne. 0) then
         call nx_perror("nx_empty_nodes()")
         stop
      end if

      do 2, i = 1, nempty
         print *, empty(i)
2     continue

      call free(ptr)
```

The following C program fragment prints the node numbers of all nodes in the root partition that failed to boot:

```
#include <nx.h>

nx_nodes_t     failed;
unsigned long  nfailed;
int            i, status;

status = nx_failed_nodes(&failed, &nfailed);

if(status != 0) {
    nx_perror("nx_failed_nodes()");
    exit(1);
}

for(i = 0; i < nfailed; i++) {
    printf("%d\n", failed[i]);
}

free(failed);
```

Note the use of the `&` operator on the variables *failed* and *nfailed* in the call to **nx_failed_nodes()**.

Handling Errors

Synopsis	Description
_call()	Special version of *call* that returns error value to caller (C only).
nx_perror(*string***)**	Print an error message corresponding to the current value of *errno*.

When an error occurs in a standard OSF/1 system call, the call indicates the error in one of two ways, depending on the error. For most errors, the call returns -1 and sets the variable *errno* to a value that describes the error. For certain severe errors (such as a segmentation violation caused by an invalid pointer parameter), the call sends a signal to the calling process; this signal may result in a *core dump*, as discussed under "Core Dumps" on page 4-56.

When an error occurs in a system call whose name begins with **nx_**, it uses the same two techniques as a standard OSF/1 system call. However, when an error occurs in a system call that is not a standard OSF/1 system call and whose name does *not* begin with **nx_**, the error is handled differently: the system prints a message on the terminal and terminates the calling process. (There are exceptions; see the manual page for each call in the *Paragon™ System C Calls Reference Manual* or *Paragon™ System Fortran Calls Reference Manual* for details.) If you program in C, you can get the same behavior as the **nx_** calls by calling the *underscore version* of the call. (Fortran does not have underscore versions.)

Underscore Calls

The underscore version of a system call is the same as the standard version except that it has an underscore added to the beginning of its name. For example, **_crecv()** is the underscore version of **crecv()**. The underscore version returns -1 if the call encounters an error and 0 or a positive value if the call is successful.

If an error occurs, the underscore version also sets the system variable *errno* to indicate the cause of the error. The include file *errno.h* declares *errno* for you and defines constants for the possible *errno* values. For example, if **crecv()** receives a message that is larger than the size specified by its *len* parameter, an error message appears and the application terminates. If you use **_crecv()** instead, this does not occur; instead, the call to **_crecv()** returns -1 and the variable *errno* is set to the value **EQMSGLONG**.

There is a standard error message for each value of *errno*, which you can print out by calling **nx_perror()**. **nx_perror()** prints its argument (any string), the current node number and process type, and the error message associated with the current value of *errno* to the standard error output in the following format:

```
(node n, ptype p) string: error_message
```

Suppose you have a program where the user can specify the size of a certain buffer with a command-line argument. If a message is received that is too long for this buffer, you would like to be able to tell the user what happened and suggest that they increase the buffer size. The following example uses the underscore version of **crecv()** to do this:

```
#include <nx.h>
#include <errno.h>

char *transbuf;
int transbuf_size;
    •
    •
    •
if(_crecv(1, transbuf, transbuf_size) == -1) {
    if(errno == EQMSGLONG) {
        /* received message too long for buffer */
        printf("Message exceeded transit buffer size!\n");
        printf("Use -t to specify a larger transit buffer.\n");
        exit(1);
    } else {
        /* some other error, print a standard error
           message and exit*/
        nx_perror("crecv");
        exit(1);
    }
}
```

Core Dumps

When an application that is executing encounters an unrecoverable error (fault), it usually results in a core dump. The application is immediately terminated and its memory contents is dumped (written) to a file named *core*. When a parallel application terminates with unrecoverable error (fault), a core directory is created instead. In either case, the contents of core dump can be examined for information on where the problem occurred in the code and give clues about what caused the unrecoverable error (fault).

See **core(1)** for information about how to control the creation, location, and contents of a core file or directory. See **signal(4)** for a information about the errors that result in a core dump.

See the *Paragon™ System Interactive Parallel Debugger Reference Manual* for information about how to examine and debug applications using core dumps with IPD.

Getting Information About Core Files

Command Synopsis	Description
coreinfo [*corename*]	Displays summary information about a core file or the core files located in a core-file directory.

The **coreinfo** command displays summary information about the contents of a core file or directory. If you use no arguments, **coreinfo** looks for a file or directory named *core* in the current working directory.

If you set the environment variable *CORE_PATH*, the command looks at the *CORE_PATH* pathname for core file or directory. If the core file or directory has been renamed or is, specify the pathname when invoking **coreinfo**. See the **core(1)** for more information on *CORE_PATH.*

The **coreinfo** commands displays the following information for an application which faulted:

- The time each process terminated ordered chronologically in the Date/Time column.
- The process ID (PID) of the faulting process,
- The node number on which the process ran.
- The process type of the process.
- The signal that terminated the process.
- The memory location where the fault occurred.
- The type of core file. This can be FULL or TRACE.
- The name of the executable.

The following example shows a **coreinfo** display of the contents of a core directory:

```
% coreinfo
Summary information for directory: /home/joe/core
Number of nodes: 3

Date/Time    Pid     Node Ptype Signal  Location   Type  Executable
---------    ---     ---- ----- ------  --------   ----  ----------
Mar 02 19:15 1049578 0    0     SIGSEGV 0x0001089c FULL  /home/joe/myapp
Mar 02 19:15 1311615 1    0     SIGSEGV 0x0001089c FULL  /home/joe/myapp
Mar 02 19:15 144560  2    0     SIGKILL 0x6004812c TRACE /home/joe/myapp
```

The core directory is */home/jöe/core*. The application that faulted and caused the core dump was executing on 3 nodes. The processes running on nodes 0 and 1 encountered a segmentation fault (SIGSEGV) which means the application attempted to access an illegal address. The instruction where the fault occurred was located at 0x0001089c. A FULL core dump was written. This means that both data and stack information were saved in the core files for these processes. The process running on node 2 was killed by the system (SIGKILL) as a consequence of the fault that occurred on the other processes. The process was executing the instruction at 0x6004812c when the kill signal arrived. The TRACE type indicates that only stack information was saved in the core file for this non-faulting process.

If a core file is for a non-parallel application, the node and process type (**Ptype** field) information would be omitted and there would be one process in the summary table only.

To translate the address given in the **Location** field to a routine name either let IPD do it for you or get a sorted list of the starting address for each function in the executable as follows:

```
% nm -hexp myapp | sort > myapp.nmlist
```

You can search this list to find which routine the address falls within.

See **coreinfo(1)** for more information about this command.

Using IPD to Examine Core

The Paragon system interactive parallel debugger (IPD) provides the following features to examine core dump information:

- The **coreload** command for loading core files.
- Symbolic information about where the fault occurred in the program.
- Stack (or frame) traceback which lists the calling sequence in the program before the fault occurred.
- Register values can be displayed.
- If the application was compiled for debug, line number and variable data can be displayed. The contents of variables can only be displayed if the type of a core file is FULL.

See the *Paragon™ System Application Tools User's Guide* and the *Paragon™ System Interactive Parallel Debugger Reference Manual* for complete information about using IPD to examine core files.

Overriding the Defaults for Core Dumps

By default, a core file is written for the first process that faults in a parallel application. You can change this by setting the following environment variables:

- *CORE_ACTION_FIRST.*
- *CORE_ACTION_FAULT.*
- *CORE_ACTION_OTHER.*

See **core(4)** for more information about how to use these environment variables to generate core files.

When certain severe errors occur in a system call, the operating system sends a signal to the calling process. The default action for certain signals is to cause a core dump. You can prevent the default action by establishing a *signal handler* for the desired signal. See **signal(4)** in the *OSF/1 Programmer's Reference* for information about signals and signal handlers.

Controlling Floating-Point Behavior

Synopsis	Description
isnan(*dsrc***)**	Determine if a **double** value is Not-a-Number (C only).
isnand(*dsrc***)**	Determine if a **double** value is Not-a-Number (C only).
isnanf(*fsrc***)**	Determine if a **float** value is Not-a-Number (C only).
fpgetround()	Get the floating-point rounding mode for the calling process (C only).
fpsetround(*rnd_dir***)**	Set the floating-point rounding mode for the calling process (C only).
fpgetmask()	Get the floating-point exception mask for the calling process (C only).
fpsetmask(*mask***)**	Set the floating-point exception mask for the calling process.
fpgetsticky()	Get the floating-point exception sticky flags for the calling process (C only).
fpsetsticky(*sticky***)**	Set the floating-point exception sticky flags for the calling process (C only).

The operating system supports a series of floating-point control calls compatible with those of UNIX System V.

NOTE

Only **fpsetmask()** is available to Fortran programs. The other floating-point control calls are available only to C programs.

Detecting Not-a-Number

The calls **isnan()**, **isnand()**, and **isnanf()** are used to determine whether a floating-point value is an IEEE NaN, or "Not-a-Number." This value can be generated as a result of certain floating-point mathematical operations and system calls, when the operands are invalid or out of range. **isnan()** and

isnand() take an argument of type **double**, and **isnanf()** takes an argument of type **float**. (**isnan()** and **isnand()** are identical except for the name.) All three calls return 1 if the argument is a NaN, and 0 otherwise.

NOTE

These calls never generate an exception, even if the argument is a NaN.

Controlling Floating-Point Behavior

The calls **fpgetround()**, **fpsetround()**, **fpgetmask()**, **fpsetmask()**, **fpgetsticky()**, and **fpsetsticky()** get and set the i860 microprocessor's floating-point control registers. The values of these registers are part of the process, and are saved and restored when the process is swapped in and out.

The **get** calls simply return the current value of the specified register for the calling process; the **set** calls set the register to the specified value for the calling process and return its previous value.

Rounding Mode

fpgetround() and **fpsetround()** get and set the i860 microprocessor's *floating-point rounding mode*, which determines what happens when a floating-point value generated in a calculation cannot be represented exactly.

The i860 microprocessor has four rounding modes:

FP_RN	Round to nearest representable number (if two representable numbers are equidistant, round to the even one).
FP_RM	Round toward minus infinity.
FP_RP	Round toward plus infinity.
FP_RZ	Round toward zero (truncate).

These symbolic names are the values of the **enum** type **fp_rnd**, which is declared in <*ieeefp.h*>. The argument of **fpsetround()** and the return values of **fpsetround()** and **fpgetround()** are of this type.

NOTE

When you convert a floating-point value to an integer type in C, it always truncates. The processor's rounding mode is ignored.

Exception Mask and Sticky Flags

fpgetsticky() and **fpsetsticky**() get and set the i860 microprocessor's *floating-point exception sticky flags*, and **fpgetmask**() and **fpsetmask**() get and set the *floating-point exception mask.*

The i860 microprocessor defines five floating-point exceptions:

FP_X_INV	Invalid operation exception.
FP_X_DZ	Divide-by-zero exception.
FP_X_OFL	Overflow exception.
FP_X_UFL	Underflow exception.
FP_X_IMP	Imprecise (loss of precision) exception.

These symbolic names are the values of the **enum** type **fp_except**, which is declared in <*ieeefp.h*>. The arguments of **fpsetsticky()** and **fpsetmask**() and the return values of **fpgetsticky**(), **fpsetsticky**(), **fpgetmask**(), and **fpsetmask()** are of this type.

The i860 microprocessor has five *exception sticky flags* and five *exception mask bits* corresponding to the five exception types. When a floating-point exception occurs, the corresponding exception sticky flag is set to 1. The corresponding exception mask bit is then examined; if it is set to 1, the exception is *trapped* and the appropriate trap handler is called.

NOTE

After an exception, you must clear the corresponding sticky flag to recover from the trap and proceed.

If the sticky flag is not cleared before the next floating-point exception occurs, an incorrect exception type may be signaled. For the same reason, when you call **fpsetmask**(), you must be sure that the sticky flag corresponding to each exception being enabled is cleared.

NOTE

fpsetsticky() and **fpsetmask()** set the sticky flags and exception mask to the specified values. Any bits not set in the call's argument are cleared.

To set or clear a particular mask or sticky flag, get the current mask or sticky flags, modify it, and then call **fpsetsticky**() or **fpsetmask**() with the modified mask or sticky flags.

Fortran Exception Mask Values

Only the **fpsetmask**() call is supported in Fortran. You use the following numeric values with **fpsetmask**():

0	No exceptions.
1	Invalid operation exception.
2	Divide-by-zero exception.
4	Overflow exception.
8	Underflow exception.
16	Imprecise (loss of precision) exception.

The argument and return value of **fpsetmask**() are integers whose values are the sum of some, none, of all of these values.

Miscellaneous Calls

Synopsis	Description
flick()	Temporarily relinquish the CPU to another process.
dclock()	Return time in seconds since booting the system.

Temporarily Releasing Control of the Processor

The **flick()** call temporarily releases control of the node processor to another process in the same application. If there are no other processes in the same application when a process calls **flick()**, control returns to the operating system. For example, if your node program has set up a number of **hrecv()**'s and has nothing else to do, it should issue **flick()**. The operating system can then more efficiently respond to an incoming message and wake up your process.

flick() does not have any effect on rollin and rollout of the application (see "Gang Scheduling" on page 2-42 for information on rollin and rollout).

Timing Execution

dclock() returns the time in seconds since the system was last booted, as a double precision number. This time is obtained from the RPM global clock and is the same on every node.

Use **dclock()** to return a relative value that you can use to measure execution time. To time an interval in your program, first obtain an initial value. Then obtain a final value and take the difference. The actual values returned by the two **dclock()** calls are not important.

Here is an example that shows how to use **dclock()** to time the execution of an iteration loop:

```
/* C version */
double start_time, end_time, diff_time;
start_time = dclock();
for(i=0;i<imax;i++) {
        •
        •
        •
}
end_time = dclock();
diff_time = end_time - start_time;
printf("Timing = %e\n", diff_time);
```

```
c Fortran version
      double precision start_time, end_time, diff_time
      start_time = dclock()
      do 100 i=1, imax
         •
         •
         •
100   continue
      end_time = dclock()
      diff_time = end_time - start_time
      write(*, 10) diff_time
10    format('diff_time = ', D15.9)
```

iPSC® and Touchstone DELTA Compatibility Calls

Synopsis	Description
flushmsg(*typesel*, *nodesel*, *ptypesel*)	Flush specified messages from the system.
ginv(*j*)	Return the position of an element in the binary-reflected gray code sequence. Inverse of **gray**().
gray(*j*)	Return the binary-reflected gray code for an integer.
hwclock(*hwtime*)	Place the current value of the hardware counter into a 64-bit unsigned integer variable.
infopid()	Return the process type of the process that sent a pending or received message.
killcube(*node*, *ptype*)	Terminate and clear node process(es).
killproc(*node*, *ptype*)	Terminate a node process.
led(*state*)	Does nothing; provided for compatibility only.
load(*filename*, *node*, *ptype*)	Load a node process.
mclock()	Return the time in milliseconds.
msgcancel(*mid*)	Cancel an asynchronous send or receive operation.
mypart(*rows*, *cols*)	Obtain the height and width of the rectangle of nodes allocated to the current application.
mypid()	Return the process type of the calling process.
nodedim()	Return the dimension of the current application (the number of nodes allocated to the application is $2^{dimension}$).
restrictvol(*fileID*, *nvol*, *vollist*)	Does nothing; provided for compatibility only.
flick()	Temporarily relinquish the CPU to another process.
dclock()	Return time in seconds since booting the system.

The iPSC and Touchstone DELTA compatible calls are provided for compatibility with the iPSC series of supercomputers and Touchstone DELTA system from Intel Corporation.These calls should not be used in new operating system applications. They either provide the same functionality as other system calls (for example, **mypid()** is identical to **myptype()** but uses the iPSC system terminology), or provide functionality that is not needed in the operating system (for example, **gray()** is not useful in a machine without a hypercube architecture).

These calls work the same as the corresponding calls on the iPSC or Touchstone DELTA system, with the following exceptions:

- **flushmsg()** does nothing.
- The only valid use of **killcube()** is **killcube(-1,-1)**.
- The only valid use of **killproc()** is **killproc(-1,-1)**.
- **led()** does nothing.
- **load()** must be preceded by **nx_initve...()** (it is equivalent to **nx_load()** but does not let you specify a list of nodes or find out the PIDs of the loaded processes).
- **msgcancel()** does nothing.
- If **numnodes()** is not a power of 2, **nodedim()** rounds it up to the next power of 2 and returns the dimension of a cube of that size. For example, if **numnodes()** is 7, **nodedim()** returns 3; if **numnodes()** is 9, **nodedim()** returns 4.
- **restrictvol()** does nothing. It always returns 0 (indicating success).

See your iPSC or Touchstone DELTA system documentation for more information on these calls.

Using Parallel File I/O 5

Introduction

The operating system provides two forms of parallel I/O to files:

- A special file system type called *PFS*, for *Parallel File System*, gives applications high-speed access to a large amount of disk storage. PFS file systems are optimized for simultaneous access by multiple nodes. Files in PFS file systems can be very large (up to several terabytes); the exact maximum depends on your system configuration. Access to PFS file systems also uses an I/O technique called *fast path I/O*, which gives superior performance for large I/O operations (64K bytes or more per read or write).

- Special I/O system calls, called *parallel I/O calls*, facilitate I/O from multiple nodes and permit I/O to very large files in PFS file systems. These calls can give applications better performance and more control over parallel file I/O than is offered by the standard C and Fortran file I/O features. These calls are compatible with the Concurrent File System™ (CFS™) calls provided by the iPSC® system.

A system running the operating system can have both PFS and non-PFS file systems. You can access files in PFS file systems with both parallel I/O calls and non-parallel I/O calls; you can use parallel I/O calls to access files in both PFS file systems and non-PFS file systems. In most cases you get the best performance when you use parallel I/O calls to access files in PFS file systems.

This chapter discusses both PFS file systems and parallel I/O calls. It also gives information on performing operations on tape devices in the operating system. For information on getting the best performance from PFS file systems and parallel I/O calls, see "I/O Performance" on page 8-23.

Disks and File Systems

Every Paragon supercomputer has one or more *disk devices* attached to it. These disk devices are attached to the system as *RAID subsystems*. RAID stands for Redundant Array of Inexpensive Disks. In a RAID subsystem, several hard disks are connected together into a unit that appears to the system as a single large disk drive. Files stored to a RAID subsystem are distributed, or *striped*, among the disks within it by the RAID controller hardware.

Each disk device is controlled by an *I/O node*: a compute node with an I/O connection. I/O nodes communicate with the other nodes in the system using the node-to-node message-passing network and with the disk drives using a SCSI interface (or other interface). The I/O nodes may or may not also run application processes; this is determined by your system administrator. Each I/O node can control RAID subsystem only, and the number of I/O nodes is limited only by the number of slots in the system, so the total amount of disk space that could be installed in a Paragon supercomputer is a terabyte or more.

The set of disk devices connected to the Paragon supercomputer's I/O nodes is divided into *file systems.* A file system can encompass anything from a portion of the space on one disk device to all of the space on several disk devices. A file system is made accessible by *mounting* it to a directory (this requires system administrator privileges). This directory is called the file system's *mount point.* For example, if the file system */dev/io0/rz0f* is mounted on the directory */home* (the directory */home* is the file system's mount point), whenever you write a file in */home* it is stored in the file system */dev/io0/rz0f.*

Each file system has a *type* that describes its internal structure and determines some of the operations that can be performed on it. The supported file system types are:

UFS UNIX File System, the standard file system type for OSF/1.

NFS Network File System, a file system type that represents a file system on another computer on the network.

PFS Parallel File System, a file system type that is optimized for access by parallel processes. This file system type is unique to the operating system.

This chapter discusses how PFS file systems work and how you can use the parallel I/O system calls provided by the operating system to access files in file systems of all types.

PFS File Systems and PFS Files

Internally, a file system of type PFS consists of one or more *stripe directories*. The stripe directories that make up a PFS file system are determined by the system administrator when the PFS file system is mounted.

Each stripe directory is usually the mount point of a separate UFS file system. Just as a RAID subsystem collects together several hard disks into a unit that behaves like a single large disk, a PFS file system collects together several file systems into a unit that behaves like a single large file system. A system running the operating system can have any number of PFS file systems.

The maximum storage capacity of a PFS file system is the sum of the capacities of the different file systems containing its stripe directories. For example, if a PFS file system consists of four stripe directories, each of which is the mount point of a UFS file system with a capacity of 100M bytes, the capacity of the PFS file system is 400M bytes. However, if another PFS file system also consists of four stripe directories, but two of them are directories in one UFS file system with a capacity of 100M bytes and the other two are directories in another UFS file system with a capacity of 100M bytes, the capacity of the PFS file system is only 200M bytes.

A *PFS file* is any ordinary file that is stored in a file system of type PFS. PFS files are distributed, or *striped*, across the stripe directories that make up the PFS file system. The amount of data from a PFS file that is stored in each stripe directory is determined by the PFS file system's *stripe unit*, a quantity that is set by the system administrator when the PFS file system is mounted. The maximum size of a file in a PFS file system is roughly 2G bytes times the number of file systems in the PFS file system. The exact maximum size is given by the formula $((((2G - 1) - r) \times n) + r)$, where r is $(2G - 1)$ **mod** *stripe_unit* (that is, the remainder when the largest integer multiple of the stripe unit that is less than $2G - 1$ is subtracted from $2G - 1$) and n is the number of *different* file systems containing the PFS file system's stripe directories

For example, suppose a PFS file system consists of three stripe directories and has a stripe unit of 64K bytes. When you write a 256K-byte file to this PFS file system, the first 64K bytes of the file are stored in the first stripe directory, the second 64K bytes in the second stripe directory, the third 64K bytes in the third stripe directory, and the last 64K bytes back in the first stripe directory.

Objects in PFS file systems that are not ordinary files (such as directories, symbolic links, and device special files) are not striped. These objects exist on the disk partition on which the PFS file system is mounted.

PFS Filenames and Pathnames

Filenames and pathnames in PFS file systems work the same as pathnames in UFS file systems. The maximum length of a pathname is 1024 characters; the maximum length of a single filename is 255 characters.

PFS Limitations

In the current release, PFS file systems and parallel I/O calls have the following limitations:

- PFS files cannot be accessed from a remote system via NFS.
- PFS does not support executable files. If you copy a binary file to a PFS file system and try to execute it, an "Operation not supported by this file system" error occurs.
- PFS does not support core files. If a core dump occurs while your current directory is in a PFS file system, a core file of length 0 is created.
- PFS does not support the **quotaon** or **sysacct** commands or the **mmap**() system call.
- PFS file regions cannot be locked by the **fcntl**() system call. However, you can use the **flock()** system call to lock the entire file.
- The maximum number of open files per process at any given time is 64. This includes the standard input, standard output, and standard error. This means that there is a practical maximum of 61 open files per process.

Using PFS Commands

In general, you use standard OSF/1 commands such as **ls**, **cat**, **cp**, and **mv** to manipulate files in PFS file systems. See the *OSF/1 Command Reference* for information on these commands. (Many commands do not work with files larger than 2G – 1 bytes, as described under "Using Extended Files" on page 5-36.) This section describes the additional file and file system commands provided by the operating system.

Displaying File System Attributes

Command Synopsis	Description
showfs [**-k**] [**-t** *type*] [*filesystem* | *directory*]	Display file system attributes.

The command **showfs** with no arguments lists the file systems on your system, together with information on each. For example:

```
% showfs
Mounted on         512-blks     avail  capacity   sunit sfactor
/                   1458308    719276    45%
/home               4060838   3373782     8%
/usr                2379194   1948124     9%
/home/.sdirs/vol0    598622    574464     4%
/home/.sdirs/vol1    598622    574464     4%
/home/.sdirs/vol2    598622    574464     4%
/home/.sdirs/vol3    598622    574464     4%
/pfs                2394488   2297856     4%       8192       4
    sdirs: /home/.sdirs/vol0
           /home/.sdirs/vol1
           /home/.sdirs/vol2
           /home/.sdirs/vol3
```

In this case, the system has eight file systems. The seven file systems mounted on the directories / (root), */home*, */usr*, */home/.sdirs/vol0*, */home/.sdirs/vol1*, */home/.sdirs/vol2*, and */home/.sdirs/vol3* are non-parallel file systems (type UFS or NFS); the file system mounted on the directory */pfs* is a PFS file system.

NOTE

There's nothing special about the name */pfs*; your PFS file systems can have any name. However, the rest of this chapter uses the convention that pathnames beginning with */pfs* are in a PFS file system.

The **showfs** command shows the following information for every file system:

`Mounted on` The directory where the file system is mounted (its *mount point*). If you need to know the file system's device name, use the standard OSF/1 command **mount** or **df**.

512-blks The total capacity of the file system in 512-byte disk blocks.

avail The number of disk blocks currently available.

capacity The approximate percentage of the file system's capacity currently in use.

In this example, the file system mounted on */usr* has a size of 2,379,194 512-byte disk blocks, of which 1,948,124 blocks are currently unused, so that the file system is approximately 9% full.

The **showfs** command shows the following additional information for each PFS file system:

sunit The file system's stripe unit, in bytes.

sfactor The number of stripe directories within the PFS file system.

sdirs The stripe directories (usually mount points of UFS file systems) within the PFS file system.

You can display the attributes of PFS files using the **ls -lP** command. See "Displaying File Attributes" on page 5-7 for more information.

In this example, the PFS file system mounted on */pfs* has a stripe unit of 8K bytes and consists of the four UFS file systems mounted on */home/.sdirs/vol0*, */home/.sdirs/vol1*, */home/.sdirs/vol2*, and */home/.sdirs/vol3*.

The **showfs** command accepts the following optional arguments:

-k Display capacity and available capacity in 1024-byte disk blocks instead of 512-byte disk blocks. The header "512-blks" changes to "kbytes".

-t *type* Display information about all file systems of type *type*, where *type* is any recognized file system type in lowercase (**pfs**, **ufs**, or **nfs**).

filesystem Display information about the file system whose device name is *filesystem*.

directory Display information about the file system mounted on *directory*.

The *filesystem* or *directory* argument overrides **-t** *type* if used together.

NOTE

You should use **showfs**, not **df**, to get information about the cumulative amount of free space in a PFS file system. Using the standard **df** command on a PFS file system only gives information about the single disk partition on which the PFS file system is mounted, so does not indicate how much space is actually available for file striping.

Displaying File Attributes

Command Synopsis	Description
ls [-l] [-P] [*filesystem* | *directory*]	Lists and generates PFS information about files.

The **ls** command has the **-l** and **-P** switches that display the stripe attributes of PFS files. The **ls -l** command displays the mode, number of links, owner, group, size, time of last modification for each file, and pathname. If the file is a special file, the size field contains the device's node number and the major and minor device numbers. For example, the following displays file information about a device special file:

```
% ls -l /dev/io0/rz0a

brw-r--r-- 1 root system  3: 3, 0 Jun 05 09:08 /dev/io0/rz0a
```

The device special file */dev/io0/rz0a* has a node number of 3 and a major/minor number of (3,0).

The **ls -P** command displays stripe attributes for PFS files as follows:

sunit — Stripe unit. The size, in bytes, of the stripe unit that is used to stripe the PFS file across the stripe directories.

sfactor — Stripe factor. The number of stripe directories. The stripe factor multiplied by the stripe unit equals the size of one PFS file stripe.

sdirs — Stripe directories. An ordered list of stripe directories in UFS or NFS file systems (typically UFS mount points) that are the storage locations for the PFS file.

You can get the stripe attributes of an individual PFS file with the **fcntl()** system call. See the **fcntl(2)** manual page for more information.

The following example displays file information about a PFS file:

```
% cd /pfs
% ls -P test

sunit 65536  sfactor 8  sdirs    /home/.sdirs/vol6    test
                                 /home/.sdirs/vol7
                                 /home/.sdirs/vol0
                                 /home/.sdirs/vol1
                                 /home/.sdirs/vol2
                                 /home/.sdirs/vol3
                                 /home/.sdirs/vol4
                                 /home/.sdirs/vol5
```

This example shows that the PFS file *test* has a stripe unit of 65536, a stripe factor of 8, and lists the eight stripe directories in the file's stripe group. The **ls** command lists the stripe directories in the order that the file data was written.

Increasing the Size of a File

Command Synopsis	Description
lsize [**-a**] *size file* [*file* ...]	Change the size of a file or files.

The **lsize** command changes the amount of disk space allocated to each specified file. You can use this command to allocate all the space you will need for a large file before you run the application that writes to the file. This makes sure that there is enough room in the file system for the file, and can also increase file I/O performance.

The **lsize** command has two forms:

lsize *size file* [*file* ...]	Sets the size of each *file* to *size* bytes.
lsize -a *size file* [*file* ...]	Increases the size of each *file* by *size* bytes.

If the specified *file* does not exist, it is created with the specified size. The *size* can be a simple integer to represent a number of bytes, or an integer followed by the letter **k**, **m**, or **g** to represent a number of kilobytes (1024 bytes), megabytes (1024K bytes), or gigabytes (1024M bytes).

For example, the following command sets the size of the file *mydat* to 5M bytes:

```
% lsize 5m mydat
```

The following command increases the size of the file *mydat* by 200K bytes:

```
% lsize -a 200k mydat
```

The additional space is allocated to the file from the file system, but it is not initialized (its contents are undefined).

lsize will not decrease the size of a file. If the specified size is smaller than the file's current size, the command has no effect.

Using Parallel I/O Calls

The rest of this chapter discusses the *parallel I/O calls* you can use in parallel applications to access both PFS and non-PFS files.

The term *parallel I/O calls* refers to all I/O calls that are provided by the operating system but not by standard OSF/1. These calls facilitate I/O on multiple nodes and permit I/O to very large files in PFS file systems. They are part of the library *libnx.a*, which is automatically searched when you link an application with the **-nx** switch. You can also use the switch **-lnx** to search *libnx.a* without using **-nx**. See "Compiling and Linking Applications" on page 2-5 for more information on these switches.

Most of the parallel I/O calls can only be used in programs running in the compute partition. They will not work, or will give unexpected results, if used in a program running in the service partition. See "Managing Partitions" on page 2-30 for more information on the service and compute partitions.

NOTE

The parameter *fileID* in the system call synopses in this chapter is an integer that represents an open file: a *unit* in Fortran, or a *file descriptor* in C.

A call description at the beginning of each section or subsection gives a language-independent synopsis (call name, parameter names, and brief description) of each call discussed in that section. Differences between C and Fortran are noted where applicable. See Appendix A for information on call and parameter types; see the *Paragon*™ *System C Calls Reference Manual* or the *Paragon*™ *System Fortran Calls Reference Manual* for complete information on each call.

Opening Files in Parallel

Synopsis		Description
gopen(*path*, *oflag*, *iomode* [, *perms*]) **gopen**(*unit*, *path*, *iomode*)	(C) (Fortran)	Open a file on all nodes and set its I/O mode.

To open a file for use by all the nodes in your application, call **gopen()**. You can use **gopen()** to open files in both PFS and non-PFS file systems. **gopen()** works like the standard **open()** operation, with the following exceptions:

- It is a *global call*. All the nodes in the application must call **gopen()**, and all must call it with the same arguments.
- It is a *synchronizing call*. Each node blocks at the **gopen()** until all the nodes have called it.
- It sets the *I/O mode* of the file, as described under "Using I/O Modes" on page 5-14.
- When called on a large number of nodes, it offers better performance and causes less system overhead.

Note that **gopen()** must be called by *all* the nodes in the application, even those that do not actually perform any I/O. For example, suppose that your application has a "manager" node that assigns I/O work to the "worker" nodes, but does no I/O itself. If you want to use **gopen()**, all the nodes, even the manager, must open the file.

Using gopen() in C

The C version of **gopen()** opens the specified file and returns a file descriptor, like the standard OSF/1 system call **open()**. In addition to being a global synchronizing call and setting the I/O mode of the file, as discussed earlier, the C version of **gopen()** has the following differences from the standard **open()**:

- It can only be used to open an ordinary file (not a directory or a device special file).
- If an error occurs, it prints an error message and terminates the calling process.

gopen() is otherwise equivalent to **open()**. For example, the following C call opens the file */pfs/mydat* for reading and writing, creating it if it does not exist, and returns a file descriptor that you can use to access it. The file's I/O mode is set to **M_GLOBAL**, and if the file is created it is given permissions 644 octal (`rw-r--r--`).

```
#include <fcntl.h>
#include <nx.h>
int fd;
fd = gopen("/pfs/mydat", O_RDWR | O_CREAT, M_GLOBAL, 0644);
```

The symbolic names for *oflag* (such as **O_CREAT**) are defined in the header file *fcntl.h*, and the symbolic names for *iomode* (such as **M_GLOBAL**) are defined in the header file *nx.h*.

See **open()** in the *OSF/1 Programmer's Reference* for information on the *oflag* parameter; see "Using I/O Modes" on page 5-14 for information on the *iomode* parameter; see **chmod()** in the *OSF/1 Programmer's Reference* for information on the *perms* parameter.

Using gopen() in Fortran

The Fortran version of **gopen()** opens the specified file for unformatted I/O on a specified unit. It is equivalent to the following Fortran **open()** statement:

```
      OPEN(unit, path, status='unknown', form='unformatted',
     x     access='sequential')
```

However, it differs from the standard Fortran **open()** in that it is a subroutine. Also, as discussed earlier, it is a global synchronizing call and sets the I/O mode of the file.

For example, the following Fortran call opens the file */pfs/mydat* on unit 10 in I/O mode **M_GLOBAL**:

```
      include 'fnx.h'
      call gopen(10, "/pfs/mydat", M_GLOBAL)
```

The symbolic names for *iomode* (such as **M_GLOBAL**) are defined in the header file *fnx.h*.

Opening Files with Standard Operations

PFS and non-PFS files can also be opened and closed with the standard OSF/1 system calls and Fortran routines. For example, to open the file */pfs/mydat* for read and write access:

```
/* C version */
fd = open("/pfs/mydat", O_CREAT | O_RDWR, 0644);

c     Fortran version
      open(unit=10, file = '/pfs/mydat',
     x     status = 'new', form='unformatted')
```

Use this method when not all nodes open the same file at the same time, or when source compatibility with other systems is necessary. (Note that, if you want to use any synchronizing calls, all nodes must open the file.)

NOTE

In Fortran, you must open the file with **form='unformatted'** to use any parallel I/O calls on the file.

The following section discusses additional special considerations for Fortran.

Special Considerations for Fortran

This section describes the special considerations that apply when you open files with the standard Fortran **open()** instead of **gopen()**.

Formatted Versus Unformatted I/O

If you call **open()** with **form='formatted'** (the default):

- You must use only Fortran I/O statements to access the file. You cannot use any of the parallel I/O calls described in this chapter on the file.

- Only one node may perform I/O to the file. If you perform formatted I/O to the same file from multiple nodes, the results are undefined.

If you open a file with **form='unformatted'**, you can use either Fortran I/O statements or parallel I/O calls to access the file. However, you must pick either one or the other: mixing Fortran I/O and parallel I/O to the same file can give unexpected results.

For the best I/O performance, you should use **gopen()**, or **open()** with **form='unformatted'**, and use parallel I/O calls for all file I/O.

If compatibility with other programs that use formatted I/O is required, you can perform formatted I/O to an internal file or a string and then use **cwrite()** to write the data to a file. However, if you use a string you must add a newline (ASCII character 10) to the end of the string using the function **char()**, since neither formatted I/O to a string nor **cwrite()** will add these for you. For example:

```
      include 'fnx.h'
      character*20 msgbuffer

      write(msgbuffer, 26) answer, char(10)
26    format('The answer is: ', i4, a1)
      call cwrite(10, msgbuffer, 20)
```

Alternatively, you can write a small program that translates your data files from unformatted to formatted and vice versa, and run it only when you need to share data with other programs.

New Files

If you call **open()** with **status='new'**, the result depends on whether or not the program is running on multiple nodes:

- If the program is running on one node (**numnodes()** is 1 or undefined), the **open()** fails if the file exists, as specified by the ANSI standard.
- If the program is running on multiple nodes (**numnodes()** is greater than 1), the file is truncated if it exists, as though you had specified **status='unknown'**.

This change makes it possible to specify **status='new'** when multiple nodes are opening a file that does not yet exist; with the standard Fortran semantics for **status='new'**, the first node to execute the **open()** statement would create the file, and the other nodes would fail because the file already exists. You can use the system call **stat()** to determine if a file exists before you open it.

Unnamed Files

If you call **open()** with no filename, the result depends on whether or not you specified **status='scratch'**:

- If you did not specify **status='scratch'**, the file is created in the current working directory with the filename *fort.nnn*, where *nnn* is the unit number. The file remains after the program terminates.
- If you specified **status='scratch'**, the file is created in the directory */usr/tmp* with the filename *FTNxxxxxxxx.nn*, where *xxxxxxxx* is the OSF/1 process ID of the creating process and *nn* is the unit number. The file does not remain after the program terminates, whether it terminated normally or abnormally.

For compatibility with the iPSC system, if you specified **status='scratch'** and the directory specified by the variable *CFS_MOUNT* exists (or, if *CFS_MOUNT* is not defined, if the directory */cfs* exists), the file *FTNxxxxxxxx.nn* is created in *$CFS_MOUNT* (or */cfs*) instead of */usr/tmp*.

Using I/O Modes

Synopsis	Description
setiomode(*fileID*, *iomode*)	Set the I/O mode for a file.
iomode(*fileID*)	Return the current I/O mode for a file.

A parallel application accesses a file with an I/O mode. You can specify a file's I/O mode when you open it with **gopen()**, and you can use **setiomode()** to change the I/O mode of a file that is already open. You can use **iomode()** to determine an open file's current mode.

Like **gopen()**, **setiomode()** is a global synchronizing call. When a node calls **setiomode()**, it blocks until all the other nodes in the application call **setiomode()** with the same arguments. **setiomode()** must be called by *all* the nodes in the application, even those that do not actually perform any I/O (this means that all nodes must open the file). Also, **setiomode()** can only be used on an ordinary file, not a directory or a device special file.

A file's I/O mode actually belongs to the file descriptor or unit through which the file is accessed, not to the file itself. The I/O mode is not stored with the file, and different programs can access the same file with different I/O modes (even at the same time).

A file's I/O mode is *not* inherited across a **fork()** (after a **fork()** all files in the child process have I/O mode **M_UNIX**).

There are six I/O modes, each of which has a name and a number:

M_UNIX (0) In this mode, each node has its own file pointer and file operations are performed on a first-come, first-served basis. If you open a file with the C **open()** call or Fortran **open** statement, it is opened with this mode (but you can change it with **setiomode()**).

M_LOG (1) In this mode, all nodes share the same file pointer and file operations are performed on a first-come, first-served basis.

M_SYNC (2) In this mode, all nodes share the same file pointer and file operations are performed in order by node number. Records may be of variable length.

M_RECORD (3)

In this mode, each node has its own file pointer and file operations are performed on a first-come, first-served basis. However, records are stored in the file in order by node number. Records must be of a fixed length.

M_GLOBAL (4)

In this mode, all nodes share the same file pointer and must perform the same file operations at the same time. All file operations are performed by a single node, which then distributes the results to the other nodes over the internode network.

M_ASYNC (5)

In this mode, each node has its own file pointer; file access is unrestricted. Atomic I/O operations are *not* preserved. This allows multiple readers/writers and variable length records.

The names **M_UNIX**, **M_LOG**, **M_SYNC**, **M_RECORD**, **M_GLOBAL**, and **M_ASYNC** are constants defined in the header files *nx.h* (for C) and *fnx.h* (for Fortran). You can use either these names or the corresponding numbers in your programs (using the names is recommended).

The I/O mode you choose for a file determines which, if any, parallel I/O calls become *synchronizing operations* (that is, each node blocks until all nodes have made the call). The synchronizing operations for each mode are described in the following sections and are summarized under "Synchronization Summary" on page 5-54.

Default I/O Mode

When you open a file with the standard **open()** call or statement, the file is opened with a default I/O mode. For non-PFS files, **M_UNIX** is always the default I/O mode. For PFS files, **M_UNIX** is the default I/O mode unless the bootmagic string *PFS_ASYNC_DFLT* is set to 1. If *PFS_ASYNC_DFLT* is set to 1, **M_ASYNC** is the default I/O mode. The value of *PFS_ASYNC_DFLT* is set by your system administrator when the system is booted. To open a file with an I/O mode other than the default, you can use **gopen()** or you can use **open()** followed by **setiomode()**.

You can find the value of the bootmagic string *PFS_ASYNC_DFLT* with the **getmagic** command, as follows:

```
% /sbin/getmagic PFS_ASYNC_DFLT
```

If this command returns 0 (zero) or no value, **M_UNIX** is the default I/O mode. If this command returns 1, **M_ASYNC** is the default I/O mode. See the **getmagic** manual page in the *Paragon™ System Commands Reference Manual* for more information about displaying values of bootmagic strings.

M_UNIX

In mode **M_UNIX** (mode number 0), each node maintains its own file pointer. File access requests are honored on a first-come, first-served basis. If two nodes write to the same place in the file, the second node overwrites the data written by the first node.

Use this mode in applications where each node performs I/O on disjoint segments of the file, or where I/O accesses are synchronized by other means (such as message-passing inherent to the application).

M_LOG

In mode **M_LOG** (mode number 1), all nodes share a single file pointer. File accesses are performed on a first-come, first-served basis. Whenever any node reads, writes, or moves the pointer, it affects the pointer position for all nodes. This may change the results of subsequent reads, writes, or moves by other nodes. This mode is useful for parallel log files.

Closing a file in this mode is a synchronizing operation. When a node closes a file, the operation blocks until all the other nodes also close the file.

M_SYNC

In mode **M_SYNC** (mode number 2), all nodes share a single file pointer and the nodes access the file in a synchronized round-robin fashion. This mode has the following characteristics:

- All nodes share a single file pointer, as for **M_LOG**.

- All the nodes in the application must open the file, and all must perform the same operations on the file in the same order. Reads and writes can be of variable sizes.

- All file operations are synchronizing. Closing, reading, writing, seeking, and detecting end-of-file (using **iseof()**) become synchronizing operations. These operations block until all nodes have called them. For example, when a node reads from a file with the parallel I/O call **cread()**, the node blocks and the read request is not honored until all other nodes have called **cread()**.

- All reads and writes to the file are performed in order by node number. For example, suppose node 3 in an application running on four nodes writes to a file with the parallel I/O call **cwrite()** before any of the other nodes. The node blocks until all the other nodes have called **cwrite()**. When all nodes have called **cwrite()**, the data from node 0 is written to the file first, followed by the data from node 1, then the data from node 2, and finally the data from node 3.

- The only valid use for **lseek()** is for all nodes to seek to the same position in the file. If nodes attempt to seek to different positions, an error occurs.

M_RECORD

Mode **M_RECORD** (mode number 3) gives results that are similar to **M_SYNC**, but it operates more efficiently. However, **M_RECORD** requires a fixed record size. This mode has the following characteristics:

- Each node has its own file pointer, as for **M_UNIX**.
- All the nodes in the application must open the file, and all must perform the same operations on the file in the same order, as for **M_SYNC**.
- Corresponding reads and writes must be of the same size on all nodes.

  When a node reads or writes to the file for the *n*th time, it must read or write the same number of bytes as the *n*th read or write by every other node. For example, if node 0 writes 100 bytes to the file with its first call to **cwrite()** and 50 bytes with its second call to **cwrite()**, then all nodes must write 100 bytes with their first call to **cwrite()** and 50 bytes with their second call to **cwrite()**.

  ## NOTE

  No verification is performed. You must make sure that all the nodes in the application make the same calls and read and write the same number of bytes.

  If different nodes read different amounts of data, incorrect data will be read. If different nodes write different amounts of data, the output of different nodes will overwrite each other and/or leave areas of the file with uninitialized data.

- All reads and writes *appear* to be performed in order by node number.

  Because reads and writes are of a known length, the operating system on each node can determine where in the file it should be reading from or writing to independently of the other nodes. The results of reading or writing a file with **M_RECORD** are the same as **M_SYNC**, but **M_RECORD** is more efficient because no synchronization is required. No seeking is required by the application; the file system automatically reads or writes file data to or from the proper offset in the file.

  For example, suppose node 2 in an application running on four nodes writes a 10-byte record. Node 2's file pointer is first moved forward by 20 bytes to leave room for the records from nodes 0 and 1. Next, node 2's record is written to the file (which advances the file pointer by 10 bytes). Finally, node 2's file pointer is moved forward by 10 bytes to leave room for node 3's record. The other nodes can fill in their "slots" at any time (earlier or later); no synchronization or communication between nodes is required.

- Closing a file is a synchronizing operation, as for **M_LOG** and **M_SYNC**.
- As for **M_SYNC**, **lseek()** becomes a synchronizing call, and the only valid use for **lseek()** is for all nodes to seek to the same position in the file. If nodes attempt to seek to different positions, an error occurs.

NOTE

When **M_RECORD** is set on UFS files, operations on these files are sequential not parallel operations.

For I/O operations with UFS files, **M_RECORD** behaves correctly and is functionally compatible with how this mode operates with PFS files. However, for UFS files, the parallelism in **M_RECORD** is disabled so all I/O operations are sequential.

M_GLOBAL

In mode **M_GLOBAL** (mode number 4), all nodes must read and write the same data to the same parts of the file at the same time. This mode gives excellent performance for programs that work this way, such as a program where every node reads in the entire contents of a large input file.

- All nodes share a single file pointer.
- All the nodes in the application must open the file, and all must perform the same operations on the file at the same time.
- All file operations are synchronizing.
- Corresponding reads and writes must be of the same size on all nodes.
- The only valid use for **lseek()** is for all nodes to seek to the same position in the file.
- When the nodes write to a file, only the data written by a single node is actually written. Data written by other nodes is ignored.

The way that this mode is implemented is that only a single node actually reads from and writes to the disk. After a read, that node distributes the data to the other nodes over the internode network. This eliminates the contention for the disk device that would otherwise occur when many nodes attempt to read from the same place in a file at the same time.

M_ASYNC

The mode **M_ASYNC** (mode number 5) is similar to the **M_UNIX** mode, except it does *not* support standard UNIX file sharing semantics for different processes accessing the same file.

- Each node has a unique file pointer.
- Nodes are not synchronized.
- Variable-length, unordered records.
- Multiple readers/multiple writers are allowed with no restrictions.

In this mode, your application must control parallel access to the file. This mode allows multiple readers and/or multiple writers to access the file simultaneously with no restrictions on record size or file offset.

This mode does not guarantee that I/O operations are atomic. For example, if multiple nodes write to the same area of a file at the same time, parts of the file area may contain data from one write while other parts may contain data from other writes. If a node reads from the same area of the file at this time, the returned data may consist partially of old data and partially of new data. Only data from one write is seen in areas of the file where multiple processes are writing simultaneously. All nodes are notified when the file size changes.

NOTE

I/O operations in M_ASYNC mode with UFS files are sequential operations not parallel operations.

For I/O operations with UFS files, M_ASYNC behaves correctly and is functionally compatible with how this mode operates with PFS files. However, for UFS files, the parallelism in M_ASYNC is disabled so all I/O operations are sequential.

An I/O Mode Example

This section provides a small example program (in Fortran and C) that you can compile and execute to illustrate the differences between the various I/O modes. The source for this program can be found on the Paragon supercomputer in */usr/share/examples/fortran/iomodes/iomodes.f* (Fortran version) or */usr/share/examples/c/iomodes/iomodes.c* (C version).

The example program works as follows: node 0 gets an I/O mode from the user (specified as a number), and sends it to the other nodes. Then all nodes call **gopen()** to open the file *mydat* in the current directory (which could be in either a PFS file system or a non-PFS file system) with the specified I/O mode.

Each node then writes 10 records to the file. Each record contains the time in seconds since the file was opened, to four decimal places, and the message "Hello from node *x*." Node 0 waits one second before each write to the file; the other nodes write as fast as they can (this demonstrates how writes to the file are differently synchronized in the different modes). When each node finishes writing, it writes a "done writing" message to the screen. Then it closes the file and writes a "finished" message to the screen (the two messages show that, in some modes, **close()** is a synchronizing operation).

Fortran Example

```
      program iomodes

      include 'fnx.h'

      integer nunit, mode, iam
      double precision start, now, loop_time, loop_start
      character*16 msg
      character*29 msgbuffer

      msg = 'Hello from node '
      nunit = 12
      iam = mynode()

      if(iam .eq. 0) then
         print *, 'Enter I/O mode (0, 1, 2, 3, 4, or 5):'
         read(*, 11) mode
11       format(i1)
         call csend(1, mode, 4, -1, myptype())
      else
         call crecv(1, mode, 4)
      endif

      call gopen(nunit, "mydat", mode)
      print 13, iam, iomode(nunit)
13    format('Node ', i4, ' using mode ', i1)
```

```
      start = dclock()
      do 100 i = 1, 10
c        *** if node 0, do nothing for 1.0 seconds ***
         if(iam .eq. 0) then
            loop_start = dclock()
101         loop_time = dclock() - loop_start
            if (loop_time .lt. 1.0) goto 101
         endif

c        *** all nodes now write a record to the file ***
102      now = dclock() - start
         write(msgbuffer, 14) now, msg, iam, char(10)
14       format(f7.4, a17, i4, a1)
         call cwrite(nunit, msgbuffer, 29)
100   continue

      print 15, iam
15    format('Node ', i3, ' done writing')
      close(nunit)
      print 16, iam
16    format('Node ', i3, ' finished')
      end
```

C Example

```
#include <fcntl.h>
#include <stdio.h>
#include <nx.h>

main()
{
    int    i, fd;
    double start, now;
    double loop_start, loop_cur;
    long   mode, iam;
    char   instring[40], msg[40];

    iam = mynode();

    if(iam == 0) {
        printf("Enter I/O mode (0, 1, 2, 3, 4, or 5):\n");
        gets(instring);
        sscanf(instring, "%ld", &mode);
        csend(1, &mode, sizeof(mode), -1, myptype());
    }
    else {
        crecv(1, &mode, sizeof(mode));
    }

    fd = gopen("mydat", O_WRONLY | O_CREAT | O_TRUNC, mode, 0666);
    printf("Node %d using mode %d\n", iam, iomode(fd));

    start = dclock();
    for(i=0;i<10;i++) {
        if(iam==0) {
            loop_start = dclock();
            loop_cur = loop_start;
            while(loop_cur - loop_start < 1.0) {
                loop_cur = dclock();
            }
        }
        now = dclock() - start;
        sprintf(msg, "%7.4f Hello from node %4ld\n", now, iam);
        cwrite(fd, msg, strlen(msg));
    }

    printf("Node %d done writing\n", iam);
    close(fd);
    printf("Node %d finished\n", iam);
}
```

Compiling and Running the Example

To compile this program to a parallel application, use the following **if77** or **icc** command:

```
% if77 -nx iomodes.f -o iomodes
```

or

```
% icc -nx iomodes.c -o iomodes
```

When you run the resulting application, you may find the output easier to understand if you run the example on four or fewer nodes. Use the **-sz** switch to determine the number of nodes on which the application runs (see "Controlling the Application's Execution Characteristics" on page 2-12 for information on **-sz** and other application switches).

For example, to run the application on two nodes of your default partition with I/O mode 1 (**M_LOG**):

```
% iomodes -sz 2
Enter I/O mode (0, 1, 2, 3, or 4):
1
Node 0 using mode 1
Node 1 using mode 1
Node 1 done writing
Node 0 done writing
Node 1 finished
Node 0 finished
%
```

The following example outputs came from the C version of the example, run on two nodes.

M_UNIX Output

In mode **M_UNIX**, each node has its own file pointer. Node 1 finishes right away. Node 0 waits before each write and overwrites the message from node 1. As a result, the file contains only the writes from node 0.

```
 1.0000 Hello from node    0
 2.0087 Hello from node    0
          •
          •
          •
 9.0711 Hello from node    0
10.0797 Hello from node    0
```

M_LOG Output

In mode **M_LOG**, the nodes share a common file pointer, but there is no synchronization. As in mode **M_UNIX**, node 1 finishes right away; but this time, node 0 appends its data to the file rather than overwriting the data from node 1.

```
 0.0000 Hello from node    1
 0.0382 Hello from node    1
          •
          •
          •
 0.0990 Hello from node    1
 0.1076 Hello from node    1
 1.0000 Hello from node    0
 2.0086 Hello from node    0
          •
          •
          •
 9.0712 Hello from node    0
10.0804 Hello from node    0
```

If the output file were large enough so that node 0 started before node 1 finished, the output of the two nodes would be interleaved in the middle of the file.

M_SYNC Output

In mode **M_SYNC**, the nodes share a common file pointer, and there is synchronization. Nodes 1 and 0 finish at around the same time. Because node 1 waits for node 0 on each write, the writes are interleaved within the file.

```
 1.0000 Hello from node    0
 0.0000 Hello from node    1
 2.0278 Hello from node    0
 1.1105 Hello from node    1
          •
          •
          •
 9.2262 Hello from node    0
 8.1641 Hello from node    1
10.2535 Hello from node    0
 9.1914 Hello from node    1
```

Node 0's records appear *earlier* in the file than node 1's, but the time value shown for each record from node 0 is *later* than for the corresponding record from node 1. This is because the value shown is the time at which **cwrite()** was called, but node 1's record was not actually written to the file until node 0 had written its record.

In this case, node 1 called **cwrite()** for the first time immediately after opening the file, at time 0, but the **cwrite()** blocked and the record was not written to the file until after node 0 called **cwrite()** for the first time, at time 1.0000 (1.0000 seconds after the file was opened). Node 1 then called **cwrite()** for the second time, at time 1.1105, but that **cwrite()** again blocked until after node 0 called **cwrite()** again at time 2.0278, and so on.

M_RECORD Output

In mode **M_RECORD**, the nodes access the file in round-robin fashion, but there is no lock-step synchronization. Node 1 finishes first. Then, node 0 goes into the file and fills in its data in the correct places. Because the records are of a fixed length, node 0 has no trouble doing this. The result is that the records are in the same order as in mode **M_SYNC**, but node 1 did not spend any time waiting for node 0.

```
 1.0000 Hello from node    0
 0.0000 Hello from node    1
 2.0208 Hello from node    0
 0.0505 Hello from node    1
        •
        •
        •
 9.1637 Hello from node    0
 0.1955 Hello from node    1
10.1841 Hello from node    0
 0.2158 Hello from node    1
```

Note that node 1 finished in only 0.2158 seconds, without having to wait for node 0.

M_GLOBAL Output

In mode **M_GLOBAL**, writes by all nodes but one (node 0 in this case) are ignored. As a result, the file contains only the writes from that node.

```
 1.0000 Hello from node    0
 2.0087 Hello from node    0
        •
        •
        •
 9.0711 Hello from node    0
10.0797 Hello from node    0
```

This output is the same as the output of **M_UNIX**, but the other nodes do not compete with node 0 for access to the disk, so this mode is more efficient. However, because this program uses such a small data file, the difference in execution time is probably not noticeable.

Note that **M_GLOBAL** is usually used for reading, not writing.

M_ASYNC Output

In mode **M_ASYNC**, each node has its own file pointer and each node controls when it writes the file. If there is no seek, node 0 overwrites the message from node 1. As a result, the file contains only the writes from node 0.

```
 1.0000 Hello from node    0
 2.0445 Hello from node    0
 3.0451 Hello from node    0
 4.0454 Hello from node    0
          •
          •
          •
 9.0469 Hello from node    0
10.0471 Hello from node    0
```

This output is the similar to the output of **M_UNIX**.

Reading and Writing Files in Parallel

You can read and write files with the familiar OSF/1 system calls and Fortran routines. For example, here is a Fortran code fragment that opens a file whose pathname is */pfs/mydat* and reads some data into an array called *array* using the Fortran **read** statement:

```
open(unit=10, file='/pfs/mydat', form='unformatted')
read 10, (array(j), j=1, n)
```

In addition to the usual I/O facilities, the operating system offers a series of parallel I/O calls, which are discussed in the following pages. These calls can be used on files in both PFS and non-PFS file systems.

Like the message-passing calls, the parallel I/O calls offer you the choice of *synchronous* or *asynchronous* I/O. The synchronous calls begin with **c** (for "complete") and do not return until the operation is complete. The asynchronous calls begin with **i** (for "incomplete") and return immediately; you use the call **iodone()** or **iowait()** to determine when the operation is complete.

If you program in Fortran, you should use the parallel I/O calls rather than Fortran I/O whenever you can. These calls offer better performance than the Fortran I/O routines, and you can test for the end of a file with **iseof()**. (This does not apply to C programmers; the usual C I/O calls are as efficient as their parallel I/O counterparts.) However, if you use parallel I/O calls on a file, you must not use Fortran file I/O statements on the same file (for example, you must not mix **write** and **cwrite()** on the same file).

NOTE

Parallel I/O to NFS files may give poor performance or unexpected results.

The Paragon supercomputer's disk I/O hardware and software are designed to support simultaneous access by large numbers of nodes. However, a remote NFS server may not be configured to support this level of access. If you perform large parallel I/O operations from large numbers of nodes to a file that is NFS-mounted from another computer, you may overload the network or the NFS server, resulting in poor performance or unexpected results.

Synchronous File I/O

Synopsis	Description
cread(*fileID*, *buffer*, *nbytes*)	Read from a file, waiting for completion.
cwrite(*fileID*, *buffer*, *nbytes*)	Write to a file, waiting for completion.
creadv(*fileID*, *iov*, *iovcnt*)	Read from a file to irregularly-scattered buffers, waiting for completion.
cwritev(*fileID*, *iov*, *iovcnt*)	Write to a file from irregularly-scattered buffers, waiting for completion.
readoff(*fileID*, *offset, buffer*, *nbytes*)	Read from a file at a specified offset, waiting for completion.
writeoff(*fileID*, *offset, iov*, *iovcnt*)	Write to a file at a specified offset, waiting for completion.
readvoff(*fileID*, *offset, buffer*, *nbytes*)	Read from a file at a specified offset, to irregularly-scattered buffers, waiting for completion.
writevoff(*fileID*, *offset, iov*, *iovcnt*)	Write to a file at a specified offset, from irregularly-scattered buffers, waiting for completion.

The calls **cread()**, **cwrite()**, **creadv()**, **cwritev()**, **readoff()**, **writeoff()**, **readvoff()**, and **writevoff()**, perform synchronous file I/O. **cread()**, **cwrite()**, **creadv()**, and **cwritev()** are equivalent to the standard OSF/1 calls **read()**, **write()**, **readv()**, and **writev()**, except that they follow the same naming and error-handling conventions as the operating system message-passing calls (see "Names of Send and Receive Calls" on page 3-7 for information on the operating system system call naming conventions; see "Handling Errors" on page 4-55 for information on the operating system error-handling conventions). Unlike their standard OSF/1 equivalents, these calls are available to Fortran programs (as well as C).

For example, here is a C code fragment that writes the message "Hello from node *x*" to the file */pfs/hello*:

```
fd = open("/pfs/hello", O_RDWR, 0644);
        •
        •
        •
sprintf(buffer, "Hello from node %d\n", iam);
cwrite(fd, buffer, strlen(buffer));
```

Here is a slightly more complicated example: a Fortran code fragment that opens a file whose pathname is */pfs/mydat*, seeks to a location, and reads some data using the synchronous call **cread()**. The data represents a matrix stored in rows of *n* four-byte elements. Each node reads *m* rows and performs a calculation with each row (calling the Basic Linear Algebra Subroutines routine **sdot()** to get the dot product of two vectors). Because each node seeks to a different place in the file, you must use I/O mode **M_UNIX** or **M_ASYNC**.

```
      open(unit=10, file='/pfs/mydat', form='unformatted')
      lseek(10, 4*mynode()*n*m, 0)

      do 10 i = 1, m
         call cread(10, arow, n*4)
         y(i) = sdot(n, arow, 1, xtotal, 1)
10    continue
```

Note that when you open a file in Fortran, you must open it as sequential and unformatted to be able to use **cread()** and **cwrite()**. (Sequential is the default access, but you must specify **form='unformatted'**.)

NOTE

Unlike their OSF/1 equivalents, these calls do not return the number of bytes read or written. If any error occurs, these calls print an error message and terminate the calling process.

Reading past the end of a file is considered an error, so you must be certain you know how many bytes remain in the file before you read from it. You can use **iseof()**, to detect end-of-file, after each **cread()** or **creadv()**. You can also use the following call to determine the length of a file:

```
length = lseek(unit, 0, SEEK_END)
```

This call sets the file pointer to the end of the file and returns the current position of the file pointer (that is, the file's length). You can then use **lseek(unit, 0, SEEK_SET)** to return the file pointer to the beginning of the file. (If the file might be larger than 2G – 1 bytes, use **eseek()** instead of **lseek()**; see "Manipulating Extended Files" on page 5-38 for more information.)

If you need to detect errors in reading and writing, you must program in C and use either the standard OSF/1 calls (**read()**, **write()**, **readv()**, and **writev()**, described in the *OSF/1 Programmer's Reference*) or the underscore versions of the parallel I/O calls (**_cread()**, **_cwrite()**, **_creadv()**, and **_cwritev()**, described under "Handling Errors" on page 4-55). The underscore versions do return the number of bytes read or written.

Asynchronous File I/O

Synopsis	Description
iread(*fileID*, *buffer*, *nbytes*)	Read from a file without waiting for completion.
iwrite(*fileID*, *buffer*, *nbytes*)	Write to a file without waiting for completion.
ireadv(*fileID*, *iov*, *iovcnt*)	Read from a file to irregularly-scattered buffers, without waiting for completion.
iwritev(*fileID*, *iov*, *iovcnt*)	Write to a file from irregularly-scattered buffers, without waiting for completion.
ireadoff(*fileID*, *offset, buffer*, *nbytes*)	Read from a file at a specified offset, without waiting for completion.
iwriteoff(*fileID*, *offset, iov*, *iovcnt*)	Write to a file at a specified offset, without waiting for completion.
ireadvoff(*fileID*, *offset, buffer*, *nbytes*)	Read from a file at a specified offset, to irregularly-scattered buffers, without waiting for completion.
iwritevoff(*fileID*, *offset, iov*, *iovcnt*)	Write to a file at a specified offset, from irregularly-scattered buffers, without waiting for completion.
iodone(*id*)	Determine whether an asynchronous I/O operation is complete. If complete, release the I/O ID.
iowait(*id*)	Wait for completion of an asynchronous I/O operation and release the I/O ID.
niodone(*id*)	Determine whether an asynchronous I/O operation is complete. If complete, return the number of bytes transferred.
niowait(*id*)	Wait for completion of an asynchronous I/O operation and return the number of bytes transferred.

The calls **iread()**, **iwrite()**, **ireadv()**, **iwritev()**, **ireadoff()**, **iwriteoff()**, **ireadvoff()**, and **iwritevoff()**, perform asynchronous file I/O. They work like **cread()**, **cwrite()**, **creadv()**, **cwritev()**, **readoff()**, **writeoff()**, **readvoff()**, and **writevoff()**, but they return immediately, without waiting for the read or write to complete. The asynchronous I/O calls return an I/O ID much like the message ID returned by the asynchronous message passing calls. You can pass this I/O ID to **iodone()** or **iowait()** to determine when the asynchronous file I/O operation has completed.

NOTE

The number of I/O IDs is limited, so you *must* use **iodone()** or **iowait()** to release each ID after you use it.

To check if an asynchronous I/O operation has completed, use the **iodone()** call. It returns 1 if the asynchronous operation has completed and 0 otherwise. You can also decide to block on the completion of an asynchronous call. Use the **iowait()** call for this. Both **iodone()** and **iowait()** take the I/O ID as an input parameter. For example (in Fortran):

```
c    Write to a file
        ioid = iwrite(12, sbuf, size)
        •
        •
        •
c    Do some calculation...
        •
        •
        •
c    Wait until the write completes
        call iowait(ioid)
```

The number of available I/O IDs is limited; be sure to release IDs that are no longer needed. There are two ways to release an I/O ID: you can issue an **iowait()**, as shown in the previous example, or you can keep issuing **iodone()**s until an **iodone()** returns 1.

NOTE

To preserve data integrity, all I/O requests that use or affect the file pointer are processed on a "first-in, first-out" basis.

This means that if an asynchronous I/O call is followed by a synchronous read, write, or seek on the same file, the synchronous call will block until the asynchronous operation has completed.

Closing Files in Parallel

It's always a good idea to close a file when you are finished using it. Whether you used **open()** or **gopen()** to open a file, and whether the file is a PFS file or a non-PFS file, you use the standard OSF/1 system calls or Fortran routines to close it.

For example, to close the file open on file descriptor *fd* (C) or unit 10 (Fortran):

```
/* C version */
close(fd);

c       Fortran version
        close(unit=10)
```

NOTE

If the I/O mode of the file being closed is anything other than **M_UNIX**, closing the file is a synchronizing operation.

See "Using I/O Modes" on page 5-14 for more information.

Detecting End-of-File and Moving the File Pointer

Synopsis	Description
iseof(*fileID***)**	Test for end-of-file.
lseek(*fileID*, *offset*, *whence***)**	Move the read/write file pointer.

The calls **iseof()** and **lseek()** are provided for both C and Fortran programmers. If you use parallel I/O calls to perform file I/O in a Fortran program, you must use **iseof()** and **lseek()** instead of the equivalent Fortran features.

The **iseof()** call returns 1 if the given file is at the end of the file and 0 otherwise. For example, the following Fortran code reads characters from the file open on unit 12, writing each one to the screen, until it reaches the end of the file:

```
      do while(iseof(12) .eq. 0)
         call cread(12, char, 1)
         print 300, iam, char
300      format('Node ', i3,' read:   ', a1)
      end do
```

The **lseek()** call moves the file pointer to *offset* bytes from the point specified by *whence*, which can be either a name or a number:

- If *whence* is **SEEK_SET**, **lseek()** moves the pointer to *offset* bytes from the beginning of the file.

- If *whence* is **SEEK_CUR**, **lseek()** moves the pointer forward *offset* bytes from its current position.

- If *whence* is **SEEK_END**, **lseek()** moves the pointer to *offset* bytes after the end of the file.

The names **SEEK_SET**, **SEEK_CUR**, and **SEEK_END** are constants defined in the header files *unistd.h* (for C) and *fnx.h* (for Fortran). For compatibility with the iPSC system, the numeric values 0, 1, and 2 are also accepted (but using the symbolic names is recommended).

lseek() returns the new position of the file pointer (measured in bytes from the beginning of the file).

For example, the following C call moves the file pointer of the file open on file descriptor *fd* to the beginning of the file:

```
#include <unistd.h>

newpos = lseek(fd, 0, SEEK_SET);
```

The following Fortran call moves the file pointer of the file open on unit 12 forward 500 bytes:

```
      include 'fnx.h'

      newpos = lseek(12, 500, SEEK_CUR)
```

NOTE

If the I/O mode of the file is **M_SYNC**, **M_RECORD**, or **M_GLOBAL**, seeking is a synchronizing operation.

See "Using I/O Modes" on page 5-14 for more information.

Flushing Fortran Buffered I/O

Synopsis	Description
forceflush()	Cause all buffered I/O to be flushed if an exception occurs.
forflush(*unit*)	Flush all buffered I/O on a particular unit.

The subroutines **forceflush**() and **forflush**() let Fortran programmers make sure that buffered I/O actually goes to the associated file or device. These subroutines are not available to C programs.

Fortran I/O to files and devices other than the user's terminal is *buffered*—that is, when you write to a file, the data is stored in a memory buffer, and only written to the corresponding file or device when the buffer is full. However, if another node is waiting for some data to appear in a file, you might want to force the contents of a unit's buffer to be written immediately. You can do this by calling **forflush**() on the unit. For example, to flush all buffered I/O on unit 9 to the corresponding file or device:

```
call forflush(9)
```

Another possible problem with buffered I/O is that if the program is interrupted by an exception, buffered data that has not yet been written to the file is lost. The subroutine **forceflush**() establishes a signal handler that flushes all buffered I/O in case of an exception. You call it as follows:

```
call forceflush
```

Note that you must call **forceflush**() *before* the exception occurs. You can use **fpsetmask**() (described under "Controlling Floating-Point Behavior" on page 4-60) to control whether or not an exception occurs in case of certain floating-point errors.

Fortran I/O to the user's terminal is not buffered. You can avoid buffering to files and devices by using parallel file I/O calls such as **cwrite**() and **iwrite**() instead of Fortran I/O. These calls do not buffer I/O into the Fortran I/O memory buffer; when the call returns, you can be sure the data has been sent to the specified file or device. (However, there may be some buffering within the operating system, which cannot be avoided.)

Using "###" Filenames

If you perform certain standard file operations on a file that contains three or more consecutive # symbols in its filename, the series of # symbols is automatically replaced by the node number (within the application) of the node that opens the file. The following C calls support "###" filenames:

access() **chdir()** **chmod()** **chown()**
creat() **mkdir()** **mknod()** **open()**
readlink() **rmdir()** **stat()** **statfs()**
truncate() **unlink()** **utimes()** **link()**
rename() **symlink(**

The Fortran calls that are equivalents of these C calls also support "###" filenames. Note that **gopen()** does *not* appear in this list.

For example, assume that you have the same program running on all the nodes of your application, and each node calls **open()** to open a file called *file###*. The result is that each node opens a separate file. Node 0 opens *file000*, node 1 opens *file001*, node 2 opens *file002*, and so on. If an application opens *file###* for reading, the specified files (*file000*, *file001*, *file002*, and so on) must exist.

If the number of digits in a node number is less than the number of # symbols in the filename, the node number is padded with zeros to the length of the sequence of # symbols. If the number of digits in a node number exceeds the number of # symbols in the filename, the filename is extended, but only when necessary. For example, calling **unlink()** on the file *data.###* in every node of an application running on 2000 nodes unlinks files *data.000*, *data.001*, *data.002* ... *data.998*, *data.999*, *data.1000*, *data.1001* ... *data.1998*, and *data.1999*.

If you use a "###" filename in a non-parallel program running in the service partition, it uses node number 0. For example, opening a file called *file###* from a service node opens *file000*. Note that this also affects standard commands that make these calls; for example, since the **rm** command calls **unlink()**, the command **rm file###** will attempt to remove the file *file000*.

Filenames containing a sequence of one or two # symbols are not affected. For example, the file *file##* is a single file that is accessible by each node.

There is nothing special about files created in this way; each file created is a single ordinary file. For example, suppose an application uses **open()** or **creat()** to create *###myfile*, writes into it, and then closes the file. This creates a series of files called *000myfile*, *001myfile*, *002myfile*, and so on. Each of these files is an ordinary file; for example, you can delete one without affecting the others, and there's nothing to prevent node 1 from opening *005myfile*.

Increasing the Size of a File

Synopsis	Description
lsize(*fileID*, *offset*, *whence***)**	Increase size of a file.

You can allocate more space to a file with **lsize()**. The **lsize()** call sets the file's size as specified by *offset* and *whence*:

- If *whence* is **SIZE_SET**, **lsize()** sets the file's size to *offset* bytes.
- If *whence* is **SIZE_CUR**, **lsize()** sets the file's size to the current file pointer position plus *offset* bytes.
- If *whence* is **SIZE_END**, **lsize()** increases the file's size by *offset* bytes.

The names **SIZE_SET**, **SIZE_CUR**, and **SIZE_END** are constants defined in the header files *nx.h* (for C) and *fnx.h* (for Fortran). For compatibility with the iPSC system, the numeric values 0, 1, and 2 are also accepted (but using the symbolic names is recommended).

For example, the following Fortran call increases the size of the file open on *unit1* to one million bytes:

```
      include 'fnx.h'

      size = lsize(unit1, 1000000, SIZE_SET)
```

The following C call increases the size of the file open on file descriptor *fd* by 500,000 bytes:

```
#include <unistd.h>
#include <nx.h>
int size, fd;

size = lsize(fd, 500000, SIZE_CUR)
```

The additional space is allocated to the file from the file system, but it is not initialized (its contents are undefined).

lsize() will not decrease the size of a file. If the size specified by *offset* and *whence* is smaller than the file's current size, the call has no effect.

The major use of this call is to ensure that enough disk space is available before you begin a lengthy calculation. Pre-allocating disk space can also improve disk performance.

Using Extended Files

A PFS file greater than or equal to 2G bytes in size is called an *extended file*. These files are stored in the same way as non-extended PFS files. However, some of the file parameters (like the file pointer and file size) do not fit into a 32-bit integer. This means that standard OSF/1 calls and commands that use these parameters cannot be used on extended files. The following two sections list the calls and commands that do not support extended files.

OSF/1 Calls that Do Not Support Extended Files

Most OSF/1 calls, such as **read()** and **write()**, don't care how big the file is and work perfectly well on extended files. The OSF/1 calls that have problems with extended files are shown in Table 5-1.

Table 5-1. OSF/1 Calls Not Supporting Extended Files

Call	Problem
fgetpos()	Can't return an offset greater than 2G – 1 bytes.
fseek()	Can't specify an offset greater than 2G – 1 bytes.
unlocked_fseek()	Can't specify an offset greater than 2G – 1 bytes.
fsetpos()	Can't return an offset greater than 2G – 1 bytes.
fstat()	Can't be used on a file larger than 2G – 1 bytes.[1]
ftell()	Can't return an offset greater than 2G – 1 bytes.
ftruncate()	Can't specify a file size greater than 2G – 1 bytes.
lseek()	Can't specify an offset greater than 2G – 1 bytes.
lstat()	Can't be used on a file larger than 2G – 1 bytes.[1]
madvise()	Can't map a file larger than 2G – 1 bytes.
mmap()	Can't map a file larger than 2G – 1 bytes.
mprotect()	Can't map a file larger than 2G – 1 bytes.
msync()	Can't map a file larger than 2G – 1 bytes.
munmap()	Can't map a file larger than 2G – 1 bytes.
stat()	Can't be used on a file larger than 2G – 1 bytes.[1]
truncate()	Can't specify a file size greater than 2G – 1 bytes.

[1] If you call **fstat()**, **lstat()**, or **stat()** on a file larger than 2G – 1 bytes, the call fails with the error **EFBIG**.

To manipulate extended files, the operating system provides special calls that perform the equivalent of **lseek()**, **stat()**, **fstat()**, and **lsize()** for extended files. These calls are discussed under "Manipulating Extended Files" on page 5-38.

OSF/1 Commands that Do Not Support Extended Files

Many OSF/1 commands make one or more of the system calls in Table 5-1, so do not work on extended files. The commands **cat**, **chgrp**, **chmod**, **chown**, **cp**, **df**, **diff**, **du**, **dump**, **dumpfs**, **find**, **fsdb**, **ls**, **mv**, **newfs**, **restore**, **showfs**, **tar**, **rm**, and **ufs_fsck** have been specially modified to support extended files; most other commands will fail if used on extended files. (Note that you must use the **-E** switch to archive an extended file with **tar**; see **tar** in the *Paragon*™ *System Commands Reference Manual* for more information.)

Table 5-2 shows the OSF/1 commands that are known to have problems with extended files. (This list is not guaranteed to be complete; other commands, not listed here, may also have problems.)

Table 5-2. OSF/1 Commands Not Supporting Extended Files

Command	Problem
compress	Can't compress a file larger than 2G – 1 bytes.
cpio	Can't handle files or archives larger than 2G – 1 bytes.
ed	Can't edit a file larger than 2G – 1 bytes.
ex	Can't edit a file larger than 2G – 1 bytes.
ftp	Can't copy a file larger than 2G – 1 bytes.
more	Can't display a file larger than 2G – 1 bytes.
rcp[1]	Can't copy a file larger than 2G – 1 bytes.
tail	Can't display a file larger than 2G – 1 bytes.
vi	Can't edit a file larger than 2G – 1 bytes.

[1] Note that although **rcp** cannot copy an extended file, **cp** can.

Manipulating Extended Files

Synopsis		Description
eseek(*fildes*, *offset*, *whence*) **eseek**(*unit*, *offset*, *whence*, *newpos*)	(C) (Fortran)	Move file pointer in extended file.
esize(*fildes*, *offset*, *whence*) **esize**(*unit*, *offset*, *whence*, *newsize*)	(C) (Fortran)	Increase size of extended file.
estat(*path*, *buffer*)	(C only)	Get status of extended file from pathname.
lestat(*path*, *buffer*)	(C only)	Get status of extended file or symbolic link from pathname.
festat(*fildes*, *buffer*)	(C only)	Get status of open extended file from file descriptor.

The **e...()** calls perform file operations on extended files. They do this by having parameters that are *extended integers* (a data type capable of representing integers greater than 2G – 1). You must use the calls described under "Performing Extended Arithmetic" on page 5-39 to operate on extended integers.

- The call **eseek()** is like **lseek()** (discussed under "Detecting End-of-File and Moving the File Pointer" on page 5-31), except that the *offset* parameter is an extended integer. The C version of this call is a function that returns the new position as an extended integer; the Fortran version is a subroutine that stores the new position in its fourth parameter.

- The call **esize()** is like **lsize()** (discussed under "Increasing the Size of a File" on page 5-35), except that the *offset* parameter is an extended integer. The C version of this call is a function that returns the new size as an extended integer; the Fortran version is a subroutine that stores the new size in its fourth parameter.

- The calls **estat()**, **lestat()**, and **festat()** are like the standard OSF/1 calls **stat()**, **lstat()**, and **fstat()** (described in the *OSF/1 Programmer's Reference*), except that they use a structure called *estat*, defined in *<sys/estat.h>*, which is the same as the OSF/1 *stat* structure except that the file size is an extended integer. These calls are available only in C, not in Fortran.

You *must* use these calls to manipulate extended files. However, you can also use these calls on non-PFS files and on PFS files less than 2G bytes in size. You can use these calls or the standard OSF/1 calls on PFS files less than 2G bytes in size.

Performing Extended Arithmetic

Synopsis		Description
eadd(*e1*, *e2*)	(C)	Add two extended integers.
eadd(*e1*, *e2*, *eresult*)	(Fortran)	
ecmp(*e1*, *e2*)		Compare two extended integers.
ediv(*e*, *n*)	(C)	Divide extended integer by integer.
ediv(*e*, *n*, *result*)	(Fortran)	
emod(*e*, *n*)	(C)	Give extended integer modulo an integer (remainder when *e* is divided by *n*).
emod(*e*, *n*, *result*)	(Fortran)	
emul(*e*, *n*)	(C)	Multiply extended integer by integer.
emul(*e*, *n*, *eresult*)	(Fortran)	
esub(*e1*, *e2*)	(C)	Subtract two extended integers.
esub(*e1*, *e2*, *eresult*)	(Fortran)	
etos(*e*, *s*)		Convert extended integer to string.
stoe(*s*)	(C)	Convert string to extended integer.
stoe(*s*, *e*)	(Fortran)	

The extended arithmetic calls manipulate 64-bit integers, also called *extended integers*. You use these calls to manipulate the parameters used by the parallel I/O calls described in the previous section.

Extended integers are signed 64-bit integers with values from $(2^{63} - 1)$ to -2^{63} (2^{63} is approximately 9.2×10^{18}).

- In Fortran, extended integers are stored in a two-element array of type **integer*4**.

- In C, extended integers are stored in a variable of type *esize_t*, a structure type defined in the header file <*sys/estat.h*>. (For compatibility with the iPSC system, there is also a header file <*estat.h*> that simply includes <*sys/estat.h*>.

You should always use extended arithmetic calls to operate on an extended integer, rather than access its internal structure.

Some of these calls return extended integers. The C versions of these calls return a value of type *esize_t*. However, Fortran does not allow functions to return arrays, so the Fortran versions of these calls are subroutines with an additional parameter: the result of the operation on the first two parameters is stored into the third parameter. For example, the following call adds the extended integers *e1* and *e2* and stores the result in *e_sum*:

```
/* C version */
#include <sys/estat.h>
esize_t e1, e2, e_sum;
e_sum = eadd(e1, e2);


c      Fortran version
       integer e1(2), e2(2), e_sum(2)
       call eadd(e1, e2, e_sum);
```

If you want to add an ordinary integer to an extended integer, you must create your own extended integer from the desired integer value. To create an extended integer, use **stoe()**. This call takes a string whose value is a number, and returns the corresponding numeric value as an extended integer. For example, the following code fragment adds 1 to the value of the extended integer *e1*. It does this by converting the string "1" to an extended integer with **stoe()**, storing the resulting extended integer in *e2*, and then adding *e2* to *e1* (note that in Fortran the string must be declared to be one character larger than the actual string being converted):

```
/* C version */
#include <sys/estat.h>
esize_t e1, e2, e_sum;
char *one = "1";

e2 = stoe(one);
e_sum = eadd(e1, e2);


c      Fortran version
       character*2 one
       parameter (one ='1')
       integer e1(2), e2(2), e_sum(2)

       call stoe(one, e2)
       call eadd(e1, e2, e_sum)
```

The other extended arithmetic calls allow you to subtract, multiply, divide, and find the remainder after division of extended integers. When you use **ediv()** or **emod()**, the divisor and answer must be 4-byte integers, not extended integers. Similarly, when you use **emul()**, the second argument must be a 4-byte integer, not an extended integer.

You can also compare two extended integers; **ecmp()** returns -1, 0, or 1, depending on whether the first extended integer is less than, equal to, or greater than the second.

Getting Information About PFS File Systems

Synopsis	Description
getpfsinfo(*buf***)**	Get PFS-specific information about all mounted PFS file systems.
statpfs(*path, fs_buffer, pfs_buffer, pfs_bufsize***)**	Get PFS-specific and non-PFS-specific information for the file system containing *path.*
fstatpfs(*fildes, fs_buffer, pfs_buffer, pfs_bufsize***)**	Get PFS-specific and non-PFS-specific information for the file system containing the file open on *fildes.*

You can use the functions **getpfsinfo()**, **statpfs()**, and **fstatpfs()** in C programs to get information about PFS file systems. These functions are not available in Fortran programs. See "PFS File Systems and PFS Files" on page 5-3 for more information on the concepts discussed in this section.

Getting Information About All Mounted PFS File Systems

getpfsinfo() gets information about all mounted PFS file systems. It is similar to the standard OSF/1 call **getmntinfo()**, except that instead of returning information in an array of *statfs* structures, it returns information in an array of *pfsmntinfo* structures. It allocates the memory for this array of structures, each of which describes one PFS file system, and stores a pointer to this array into its argument. **getpfsinfo()** returns the number of elements in this array. The *pfsmntinfo* structure, defined in the header file *pfs/pfs.h,* contains the following fields:

m_mntonname Directory on which the PFS file system is mounted.

m_statpfs *statpfs* structure that describes the PFS file system.

The *statpfs* structure, also defined in the header file *pfs/pfs.h,* describes the PFS-specific attributes of a file system. This is a variable-size structure. It contains the following fields:

p_reclen Total size of this *statpfs* structure, in bytes.

p_sunitsize Stripe unit size for this PFS file system, in bytes.

p_sfactor Number of stripe directories within this PFS file system.

p_sdirs List of stripe directories within this PFS file system. The number of pathnames in the list is specified by *p_sfactor.*

Each pathname in *p_sdirs* is a structure of type *pathname_t* (defined in *pfs/pfs.h*); you can use the **NEXTPATH()** macro defined in *pfs/pfs.h* to examine each pathname in turn.

Here's an example of **getpfsinfo()**:

```
#include <sys/types.h>
#include <nx.h>
#include <pfs/pfs.h>

main() {
    struct pfsmntinfo *pfsinfo;
    struct statpfs    *sattr;
    pathname_t        *sdir;
    int                cnt, i, incr;

    cnt = getpfsinfo(&pfsinfo);

    if(cnt == 0) {
        printf("No PFS file systems mounted\n");
    } else {
        for(i = 0; i < cnt; i++) {
            printf("Mount point: %s\n", pfsinfo->m_mntonname);

            sattr = &(pfsinfo->m_statpfs);
            printf("  Stripe unit size: %d\n",
                   sattr->p_sunitsize);
            printf("  Stripe factor: %d\n", sattr->p_sfactor);

            sdir = &(sattr->p_sdirs);
            printf("  Stripe directories:\n");
            for(i = 0; i < sattr->p_sfactor; i++) {
                printf("    %s\n", sdir->name);
                sdir = NEXTPATH(sdir);
            }

            incr = sizeof(pfsinfo->m_mntonname)
                   + sattr->p_reclen;
            pfsinfo = (struct pfsmntinfo *)((char *)pfsinfo
                      + incr);
        }
    }
}
```

This program prints out the attributes of all mounted PFS file systems, something like the command **showfs -t pfs**. Note that you must use the **NEXTPATH()** macro to step through the *p_sdirs* field of the *statpfs* structure, and you must increment the pointer into the array of *pfsmntinfo* structures by the size of the current *pfsmntinfo* structure (using the value of its *p_reclen* field).

Getting PFS Information About a Single File System

statpfs() gets information about a file system given the pathname of a file or directory in that file system; **fstatpfs()** gets information about a file system given the file descriptor of an open file in that file system.

These functions get both general and PFS-specific information about the specified file system. They can be used on both PFS and non-PFS file systems, but they return PFS-specific information only for PFS file systems. They are similar to the standard OSF/1 calls **statfs()** and **fstatfs()**, except that instead of returning information in a *statfs* structure, they return information in an *estatfs* structure and a *statpfs* structure.

- The *estatfs* structure, defined in the header file <*pfs/pfs.h*>, describes the basic attributes of the file system. It is just like the *statfs* structure defined in <*sys/mount.h*>, except that some of its fields are of type *esize_t* (see "Performing Extended Arithmetic" on page 5-39 for information on this type). This is necessary because some of the values returned for PFS file systems are too large to be stored into an ordinary integer.

  Some of the more generally useful fields of the *estatfs* structure are:

  f_type The type of the file system, expressed as a constant such as **MOUNT_UFS**, **MOUNT_NFS**, or **MOUNT_PFS** (these constants are defined in <*sys/mount.h*>).

  f_bavail Number of free 1024-byte disk blocks in the file system available to ordinary users, expressed as a value of type *esize_t*.

  f_mntonname Directory on which the file system is mounted, expressed as a string.

  f_mntfromname Device name of the file system, expressed as a string.

  See **statpfs()** in the *Paragon™ System C Calls Reference Manual* for a complete description of all fields in the *estatfs* structure.

- The *statpfs* structure is the same *statpfs* structure described for **getpfsinfo()** in the previous section. However, the way it is returned is different: **getpfsinfo()** allocates space for several *statpfs* structures and returns you a pointer to this space, but **statpfs()** and **fstatpfs()** store information in a *statpfs* structure that you provide.

  Because the *statpfs* structure is variable-size, you must tell **statpfs()** and **fstatpfs()** how big your *statpfs* structure is; you do this with the third parameter of **statpfs()** and **fstatpfs()** (called *pfs_bufsize*). Then you must check the *p_reclen* field in the returned *statpfs* structure to be sure the returned information fit in your provided structure; if it didn't, try again with a larger structure.

Here's an example of **statpfs()**:

```
#include <sys/types.h>
#include <sys/mount.h>
#include <malloc.h>
#include <nx.h>
#include <pfs/pfs.h>

#define SDIRS_INIT_SIZE  1024

main(int argc, char **argv) {
    struct statpfs *statpfsbuf;
    int             bufsize;
    struct estatfs  estatbuf;
    pathname_t     *sdir;
    char            blocks[80];
    int             i;

    if(argc != 2)
    {
        printf("Usage: %s <mountpoint>\n", argv[0]);
        exit(1);
    }

    bufsize=sizeof(struct statpfs) + SDIRS_INIT_SIZE;

    statpfsbuf=(struct statpfs *)malloc(bufsize);

    if(statpfs(argv[1], &estatbuf, statpfsbuf, bufsize) < 0)
    {
        nx_perror("statpfs");
        exit(1);
    }

    if(statpfsbuf->p_reclen > bufsize)
    {
        bufsize=statpfsbuf->p_reclen;
        statpfsbuf=(struct statpfs *)realloc(statpfsbuf,
                                             bufsize);

        if(statpfs(argv[1], &estatbuf, statpfsbuf, bufsize)
           < 0)
        {
            nx_perror("statpfs");
            exit(1);
        }
    }
    printf("Selected PFS statistics for %s:\n", argv[1]);

    /* From estatfs structure */

    printf("  File system type: %d\n", estatbuf.f_type);
    etos(estatbuf.f_bavail, blocks);
    printf("  # of 1K blocks available: %s\n", blocks);
    printf("  Mount point: %s\n", estatbuf.f_mntonname);
    printf("  Device name: %s\n", estatbuf.f_mntfromname);

    /* From statpfs structure */

    printf("  Stripe unit size: %d\n",
           statpfsbuf->p_sunitsize);
    printf("  Stripe factor: %d\n", statpfsbuf->p_sfactor);

    printf("  Stripe directories:\n");
    sdir = &(statpfsbuf->p_sdirs);
    for (i = 0; i < statpfsbuf->p_sfactor; i++) {
        printf("    %s\n", sdir->name);
        sdir = NEXTPATH(sdir);
    }
}
```

This program prints out the attributes of the file system containing the file specified by its first argument. Note that you must allocate enough space for the *statpfs* structure plus the stripe directory pathnames and check the returned *p_reclen* against the currently-allocated size of the structure (*bufsize*).

This example starts off by allocating an extra **SDIRS_INIT_SIZE** bytes (an arbitrary value) for the stripe directory pathnames. If *p_reclen* is larger than the size of the structure, this example uses **realloc()** to enlarge the structure and calls **statpfs()** again. It then uses the **NEXTPATH()** macro to step through the *p_sdirs* field of the *statpfs* structure, as discussed earlier for **getpfsinfo()**.

Controlling Open Files

Synopsis		Description
fcntl(*fd, request, argument*)	(C)	Controls an open file descriptor.
fcntl(*unit, request, argument*)	(Fortran)	

You can use the standard OSF/1 system call **fcntl()** to control a file that is opened with **open()** or **gopen()**. The header file *sys/fcntl.h* defines the structures and constants you need to use **fcntl()**. This call lets you to get and set stripe attributes for a PFS file. The following values for the *request* parameter let you get and set a file's stripe attributes:

F_GETSATTR Gets the PFS stripe attributes of the file associated with the *filedes* parameter. The *argument* parameter is a pointer to a *sattr* structure that returns the file's stripe attributes.

F_SETSATTR Sets the PFS stripe attributes of the file associated with the *filedes* parameter. The *argument* parameter is a pointer to a *sattr* structure that contains the file's new stripe attributes.

When using **F_SETSATTR**, you can only permanently set stripe attributes on PFS files that have not been written and are zero-length. You can use **F_SETSATTR** to temporarily set stripe attributes on PFS files that have been written and are read-only. Temporarily setting stripe attributes, affects the file descriptor only, and the changes go away when the file is closed. Temporarily setting stripe attributes can be useful when you are writing matrix data to a file in one decomposition and reading the matrix data back in a different decomposition.

You can get or set the following stripe attributes:

stripe unit size The unit of data interleaving in bytes used in the PFS file.

stripe factor The number of stripe directories the file is striped across. When setting the stripe factor, this value must be less than or equal to the current stripe factor of the PFS file.

base stripe directory
The stripe directory at which striping begins for the file. The base stripe directory must specify a subset of the PFS file's stripe group. The base stripe directory must be between 0 and (*stripe factor*-1).

Getting Stripe Attributes for Open Files

When you call **fcntl()** with **F_GETSATTR** or **F_SETSATTR** as its second argument, you must use a structure of type *sattr* as the third argument. The *sattr* structure in C is defined as follows:

```
struct sattr {
    size_t  s_sunitsize;  /* size of each stripe unit */
    uint_t  s_sfactor;    /* stripe factor */
    uint_t  s_start_sdir; /* base stripe directory */
};
```

The following example C program opens a PFS file, gets a PFS file's stripe attributes, and prints the results.

```
#include <fcntl.h>
#include <stdio.h>
#include <pfs/pfs.h>

void print(void);
struct sattr    sattr;

main()
{
     int fd;

     fd = open("/pfs/myfile", O_RDWR|O_CREAT, 0644);
     if (fd < 0) {
          perror("open");
          exit(1);
     }

     if (fcntl(fd, F_GETSATTR, &sattr) != 0 ) {
          perror("fcntl");
          exit(1);
     }

     printf("stripe attributes for /pfs/myfile: \n");
     printf("  stripe unit size = %d\n", sattr.s_sunitsize);
     printf("  stripe factor = %d\n", sattr.s_sfactor);
     printf(" base stripe dir = %d\n", sattr.s_start_sdir);

     close(fd);
}
```

Setting Stripe Attributes for Open Files

The *sattr* structure in Fortran is as follows:

```
STRUCTURE /sattr/
   INTEGER*4 s_sunitsize
   INTEGER*4 s_sfactor
   INTEGER*4 s_start_sdir
END STRUCTURE
```

The following example Fortran program opens a PFS file and sets a PFS file's stripe attributes.

```
INCLUDE 'fnx.h'

INTEGER unit

STRUCTURE /sattr/
   INTEGER*4 s_sunitsize
   INTEGER*4 s_sfactor
   INTEGER*4 s_start_sdir
END STRUCTURE

RECORD /sattr/ sattr

unit = 12
OPEN(unit, FILE='/pfs/myfile',FORM='UNFORMATTED);

sattr.s_sunitsize = 32768
sattr.s_sfactor = 3
sattr.s_start_sdir = 1

CALL fcntl(unit,F_SETSATTR, sattr)

close(unit)

END
```

Controlling Tape Devices

Synopsis	Description
ioctl(*fd*, *request*, *argp*)	Perform an operation on an open tape or other device.

You can use standard OSF/1 I/O calls or parallel I/O calls to open, read, and write tape devices. To control tape devices, use the standard OSF/1 system call **ioctl**(). The header file <*sys/mtio.h*> defines the tape-specific structures and constants you need.

NOTE

Only one node at a time can open a tape device, and it must use I/O mode **M_UNIX** (0).

The include file *sys/mtio.h* defines three constants you can use as the second argument of **ioctl**():

MTIOCTOP Perform operation on tape.

MTIOCGET Get status of tape.

MTIODISPLAY Write the display for the 3480 tape drive.

MTIODGET Returns the tape density.

MTIOPOS Returns the tape position.

The rest of this section explains the details of using these constants.

Naming Tape Devices

The operating system uses the following conventions for naming tape devices:

/dev/ioN/rmtX Raw cartridge tape, rewinds automatically when closed.

/dev/ioN/rmtXl1 Raw cartridge tape, rewinds automatically when closed. This is for a 3480 tape device only. The *l1* specifies that the device is logical unit 1.

/dev/ioN/nrmtX Raw cartridge tape, does not rewind automatically when closed.

/dev/ioN/nrmtXl1 Raw cartridge tape, does not rewind automatically when closed. This is for a 3480 tape device only. The *l1* specifies that the device is logical unit 1.

/dev/ioN/rmtcX Raw cartridge tape with compression, rewinds automatically when closed.

/dev/ioN/nrmtcX Raw cartridge tape with compression, does not rewind automatically when closed.

NOTE

The *rmtc* devices can only be used with tape drives that support data compression.

In each case, *N* is the node number of the I/O node to which the tape device is connected, and *X* is the SCSI ID of the tape device (typically 6). So, for example, to use the cartridge tape device with SCSI ID 6 on the boot node (node 0) and have it rewind automatically when closed, use the pathname */dev/io0/rmt6.* To use the same device but have it *not* rewind automatically when closed, use the pathname */dev/io0/nrmt6.*

Performing Operations on Tape Devices

When you call **ioctl()** with **MTIOCTOP** as its second argument, you must use a structure of type *mtop* as the third argument. This structure tells **ioctl()** what operation to perform. The *mtop* structure is defined as follows:

```
struct mtop {
unsigned int mt_op;    /* operation to perform */
daddr_t  mt_count;     /* how many operations to perform or */
                       /* tape density; daddr_t is a long */
};
```

The valid values of the *mt_op* field for **MTIOCTOP** include the following constants:

MTWEOF Write *mt_count* end-of-file marks.

MTFSF Space the tape forward by *mt_count* files.

MTBSF Space the tape backward by *mt_count* files.

MTFSR Space the tape forward by *mt_count* records.

MTBSR Space the tape backward by *mt_count* records.

MTREW Rewind the tape. If the tape has been written to, writes two end-of-file marks before rewinding. (Two end-of-file marks indicate the end of data.)

MTOFFL Rewind the tape and put the drive offline. If the tape has been written to, writes two end-of-file marks before rewinding.

MTCACHE Enable the drive's on-board buffer.

MTNOCACHE Disable the drive's on-board buffer.

MTLOCATE Position the tape at the requested block.

MTDENSITY Sets the tape density. The count (*mt_count*) identifies the density. It is either **EXB_85** (Exabyte 8505) or **EXB_82** (Exabyte 8200).

MTNOP No operation, sets status only.

Closing the tape device after writing to it also writes an end-of-file mark (or two end-of-file marks if the tape was opened in variable-block mode or the tape mode "rewind" is set). If the tape was opened in variable-block mode, the tape head is then positioned between the two end-of-file marks, so that any subsequent write will overwrite the second one.

For example, the following C program rewinds the tape on the device connected to */dev/io0/rmt6*:

```
#include <fcntl.h>
#include <errno.h>
#include <sys/mtio.h>

main() {
    int fd;
    struct mtop s;
    fd = open("/dev/io0/rmt6", O_RDONLY, 0666);
    if(fd == -1) {
        perror("opening /dev/io0/rmt6");
        exit(1);
    }

    s.mt_op = MTREW;
    s.mt_count = 1;
    if (ioctl(fd, MTIOCTOP, &s) == -1) {
        perror("rewinding tape");
        exit(2);
    }
}
```

If you want to make a tape on the Exabyte 8505 tape drive but write the tape so that it is readable on the Exabyte 8200 tape drive, you must issue an **ioctl()** with **MTIOCTOP** and set *mt_op* to **MTDENSITY** and *mt_count* to 0x14.

The new density takes effect when you write data at the beginning of the tape. If you change the density when the tape is not positioned at the beginning and then write data, the new density will not take effect. It you then reposition the tape at its beginning and then write data, the new density takes effect.

Getting Status of Tape Devices

When you call **ioctl()** with **MTIOCGET** or **MTIODGET** as its second argument, you must provide a structure of type *mtget* as the third argument. The *mtget* structure is defined as follows:

```
struct mtget {
   short  mt_type;  /* type of magtape device */
   short  mt_dsreg; /* ''drive status'' register */
   short  mt_erreg; /* ''error'' register */
   short  mt_resid; /* residual count */
/* the following two are not yet implemented */
   daddr_t mt_fileno;      /* file number of current position */
   daddr_t mt_blkno;      /* block number of current position */
/* end not yet implemented */
};
```

ioctl() fills in the elements of this structure with information about the device. The value of the *mt_type* field is always **0x0C** (indicating a generic SCSI device). The values of the *mt_dsreg* and *mt_erreg* fields are device-dependent.

When **MTIODGET** is the second argument, the tape density is returned in *mt_dsreg*. The value is 0x00 for the Exabyte 8505 tape drive and 0x14 for the Exabyte 8200 tape drive. Note that, if you change the density with **MTIOTCOP**, you must write something to the tape before a **MTIODGET** returns the changed tape density.

For example, the following C program prints the status of the device connected to */dev/io0/rmt6*:

```
#include <fcntl.h>
#include <errno.h>
#include <sys/mtio.h>

main() {
    int fd;
    struct mtget s;

    fd = open("/dev/io0/rmt6", O_RDONLY, 0666);
    if(fd == -1) {
            perror("opening /dev/io0/rmt6");
            exit(1);
        }

        if (ioctl(fd, MTIOCGET, &s) == -1) {
            perror("getting status of tape");
            exit(2);
        }

        printf("mt_type  = 0x%x\n", s.mt_type);
        printf("mt_dsreg = 0x%x\n", s.mt_dsreg);
        printf("mt_erreg = 0x%x\n", s.mt_erreg);
        printf("mt_resid = 0x%x\n", s.mt_resid);
    }
```

Writing the 3480 Display

When you call **ioctl()** with **MTIODISPLAY** as its second argument, you must provide a structure of type *load_display* as the third argument. The *load_display* structure is defined as follows:

```
struct load_display{
        char            message[17];
};
```

Getting the Tape Position

When you call **ioctl()** with **MTIOPOS** as its second argument, you must provide a structure of type *position* as the third argument. The *position* structure is defined as follows:

```
struct position{
        char            flags;
        char            partition;
        char            reserved_1;
        char            reserved_2;
        unsigned long   first_block;
};
```

Synchronization Summary

Table 5-3 lists the I/O modes and summarizes the I/O calls that are synchronizing calls in each one. Table 5-4 lists the most commonly-used I/O calls and summarizes the I/O modes that cause them to become synchronizing calls.

Table 5-3. Synchronization in Each I/O Mode

I/O Mode	I/O Calls that Synchronize
M_UNIX	**gopen()** and **setiomode()**
M_LOG	**gopen()**, **setiomode()** and **close()**
M_SYNC	All
M_RECORD	**gopen()**, **setiomode()**, **lseek()**, **eseek()**, and **close()**
M_GLOBAL	All
M_ASYNC	**gopen()** and **setiomode()**

Table 5-4. File I/O Calls that Synchronize

Call	I/O Modes Causing the Call to Synchronize
close()	M_LOG, M_SYNC, M_RECORD, and M_GLOBAL
cread() and **creadv()**	M_SYNC and M_GLOBAL
cwrite() and **cwritev()**	M_SYNC and M_GLOBAL
eseek()	M_SYNC, M_RECORD, and M_GLOBAL
gopen()	All
iread() and **ireadv()**	M_SYNC and M_GLOBAL
iseof()	M_SYNC and M_GLOBAL
iwrite() and **iwritev()**	M_SYNC and M_GLOBAL
lseek()	M_SYNC, M_RECORD, and M_GLOBAL
setiomode()	All

SMP Programming 6

Introduction

This chapter describes the symmetric multiprocessing programming (SMP) model on a Paragon system. This programming model is supported for both MP and GP systems.

SMP programs are multi-threaded programs that run on one or more processors. A single image of the operating system runs on these processors. Also, none of the threads has any specialized access to the hardware.

The threads are implemented as pthreads, which is short for *POSIX threads*. The implementation is based on, but is not strictly conformant to, the *POSIX Threads Extension [C language] P1003.4a/D4 (Draft 4), August 1990*, which is not the most recent one. Consequently, your pthread programs may not be portable to or from other systems.

An MP system has three processors on each compute node board. Two of those processors run application code while the third is dedicated to message passing. A GP system has two processors on each compute node board. One of those processors is dedicated to message passing.

A pthread is one of the resources of a process. The pthread is the resource that performs the actual execution of code. Other resources are memory objects and open files. On GP systems, the pthreads all run on one application processor. On MP systems, a pthread can run on either of the two application processors.

A process may have several pthreads. When it does, it has several threads of execution. When a single process has more than one pthread, each pthread executes independently, but shares resources with other threads. For example, all the pthreads in a single process share memory; when one pthread writes to a global variable in memory, it modifies the value of that variable for all threads.

On a Paragon system, there are two approaches to SMP programming. You should not use both in the same program. The two approaches are the following:

1. Let the compiler do it. Both C and Fortran compilers are parallelizing compilers. On MP systems, the compiler emits code that creates a pthread on each additional application processor. Note that currently there is only one additional processor, and so there is only one additional

pthread. When the application program encounters a loop that the compiler has accepted for parallelization, a portion of that loop is executed by the pthread. You can control the operation of the compiler with switches and directives.

2. Explicitly create your own pthreads. If you create your own pthreads, you must use the C programming language. No Fortran interface is provided. Because pthreads share resources (like memory) you must carefully manage your pthreads. Software that can be safely executed by two or more threads at the same time is referred to as *thread-safe* or *reentrant.*

NOTE

Pthreads are *not* the same as, and are *not* compatible with, cthreads or Mach threads. These other thread types are not supported. They are incompatible with *libc_r.a* and cannot be recognized or managed by the calls in *libpthreads.a.*

Libraries for SMP Programming

To use the compiler to achieve parallelization, specify the **-Mconcur** compiler switch. This switch automatically links in the libraries *libpthreads.a*, *libc_r.a*, and *libmp3.a*.

If you write your own custom pthread applications, you must link in at least *libpthreads.a* and *libc_r.a*. Do not link in *libmp3.a*; it is only for compiler use.

The libraries for SMP programming are as follows:

libpthreads.a Contains thread management calls, such as **pthread_create()**. If you are relying on the compiler to perform the parallelization, do not issue these calls directly. This library is automatically linked in when you specify the **-Mconcur** switch.

libc_r.a Contains reentrant versions of standard C library (*libc.a*) calls, such as **printf()**. This library is automatically linked in when you specify the **-Mconcur** switch. If you do not use **-Mconcur** and instead write your own custom pthreads, you must link in both *libc_r.a* and *libpthreads.a.* Linking in *libc_r.a* without also linking in *libpthreads.a* results in a link error.

libmp3.a Contains runtime support functions for the loop parallelizing compiler that manage compiler-generated pthreads. This library is automatically linked in when you specify the **-Mconcur** switch.

libm_r.a Contains reentrant versions of standard *libm.a* calls, such as **sqrt()**. Also has thread-specific *errno* numbers. This library is automatically linked in when you specify the **-Mconcur** and **-lm** switches on the compiler invocation. If you do not use **-Mconcur**, you must explicitly specify **-lm_r** to obtain the reentrant version of the math library.

libkmath_r.a Contains reentrant versions of Basic Math Library calls, such as **dgemm()**. This library is automatically linked in when you specify the **-Mconcur** and **-lkmath** switches on the compiler invocation. If you do not use **-Mconcur**, you must explicitly specify **-lkmath_r** to obtain the reentrant version of the Basic Math Llibrary.

libksignal_r.a Contains reentrant versions of Signal Processing Library calls. The Signal Processing Library is an optional product available with Paragon systems. This library is automatically linked in when you specify the **-Mconcur** and **-lksignal** switches on the compiler invocation. If you do not use **-Mconcur**, you must explicitly specify **-lksignal_r**.

Setting _REENTRANT

Whether you rely on the compiler or write your own custom pthreads, you should set the preprocessor symbol **_REENTRANT**. Do this by including the file *pthread.h* or defining **_REENTRANT** on the command line.

The symbol **_REENTRANT** is defined in *pthread.h.* Because some programs include other header files before *pthread.h*, you may want to specify the switch **-D_REENTRANT** on the command line when compiling a program that uses multiple pthreads. This symbol ensures that the correct versions of call prototypes and preprocessor symbols are pulled in from header files.

Relying on the Compiler vs. Custom Pthreads

The primary benefit of relying on the compiler for parallelization is that it is an easy way to speed up single node performance. You can achieve loop parallelization without having to learn the complicated techniques of pthread programming. Drawbacks are that only one pthread per processor is allowed and that not all loops can be parallelized.

You may be convinced you can do a better job than the compiler. The compiler is necessarily conservative about what loops it parallelizes. Because you know the details of your code, you may be able to make better decisions.

By writing your own pthreads, you can explore parallelization beyond loops. Loop parallelization is most useful for data decomposition.You may wish instead to parallelize your code at a higher level. That is, you may want to perform task decomposition.

For example, you may detect independent calculations in your application (at a higher level than loop iterations) that for clarity and performance reasons, you assign to different pthreads. For example, you may assign one pthread to exchange messages with other compute nodes and another pthread to perform calculations.

Note that as an application programmer, you do not have control over which processor runs your pthread. That is determined by the operating system kernel at runtime.

Relying on the Compiler

Using the compiler is the easiest way to achieve loop parallelization. The process is not entirely automatic. You control the operation of the compiler with compiler switches, directives (for Fortran), and pragmas (for C). Examples are shown later in this section.

For a complete description of compiler switches and directives, refer to the *Paragon™ Fortran Compiler User's Guide* and the *Paragon™ C Compiler User's Guide*. This section describes the most common use of the most common options.

Limitations when Relying on the Compiler

The pthreads package has the following limitations in the current release:

- Currently, only the following libraries are thread-safe:

  libpthreads.a
  libc_r.a
  libm_r.a
  libkmath_r.a
  libksignal_r.a

  You need not specify any of these libraries directly. The compiler switch **-Mconcur** ensures that they are linked in. It is important to note, however, that *libnx.a* (which contains the calls discussed in the Paragon system reference manuals) is not thread-safe. Hence, you must not try to parallelize loops that contain calls to *libnx.a.*

- For performance reasons, the compiler-parallelized code generates only one additional pthread for each additional application processor. This means one additional pthread when two application processors are used.

- Any global variables used or set by a non-thread-safe library may also have to be protected. In particular, you should pay attention to global variables set within your own routines when these routines appear within loops and you have allowed the parallelization of loops containing calls by specifying the compiler switch **-Mcncall.**

- The application development tools currently are not thread-aware and do not have any features to support pthreads.

  In particular, IPD does not support the debugging of threaded applications. Similarly, the debugging of core files generated by threaded applications is not supported. This includes applications that use pthreads as a result of using the **-Mconcur** switch (or the associated pragma) with the R5.0 version of the compilers.

Setting DFLT_NCPUS

If you run your application on an MP system, a pthread is created at runtime. If, for debugging purposes, you do not want to create a pthread; instead you want your application to run on only one processor, set *DFLT_NCPUS* to 1. In this case, the application may run on either of the two application processors.

On an MP system, setting *DFLT_NCPUS* to 2 is the same as not setting it. Setting *DFLT_NCPUS* to a value greater than 2 is an error that causes the application to terminate.

On a GP system, setting *DFLT_NCPUS* to 1 is the same as not setting it. Setting *DFLT_NCPUS* to a value greater than 1 is an error that causes the application to terminate.

Set the environment variable as follows:

```
% setenv DFLT_NCPUS 1
```

Compiler Switches

The primary compiler switches used for loop parallelization are **-Mconcur**, **-Mcncall**, **-Miomutex**, and **-Mreentrant**. You must use at least **-Mconcur**.

- **-Mconcur** tells the compiler to turn on parallelization. You can choose block or cyclic loop parallelization.

  The **-Mconcur** switch tells the compiler to parallelize those loops that qualify. When loops are nested, the compiler will attempt to parallelize the outermost loop. More typically, however, you would specify **-Mconcur** and then use directives or pragmas to target specific loops for parallelization.

  Loops that do not qualify for parallelization typically contain one of the following: a cross-iteration dependency, a function call, a conditional statement, a small number of iterations.

  No loops will be parallelized unless **-Mconcur** is present on the compiler invocation line. This switch must be present on both the compilation and link steps. If it only exists on the compilation step, the linker will issue undefined symbol messages for runtime support routines.

  Without options, **-Mconcur** directs the compiler to use block parallelization. This means that processor 0 and processor 1 each get a contiguous block of iterations. The iterations are balanced between the processors, and the specified number iterations is performed before either processor leaves the parallelized loop.

  You can instead specify cyclic parallelization. Do this by adding an option to **-Mconcur** as follows **-Mconcur=dist:cyclic**. Cyclic parallelization means that every other iteration is assigned to the other processor.

- **-Mcncall** tells the compiler to allow parallelization for loops that contain a function or subroutine call. If you specify **-Mcncall**, you must also specify **-Mconcur**. Also, you must ensure that any routines within loops targeted for parallelization are thread-safe. You know that routines from the C library are thread-safe because **-Mconcur** linked in the reentrant version of *libc.a*, *libc_r.a*.

  If you specify **-Mcncall** and the routine in a parallelized loop itself contains parallelized loops, those loops are executed sequentially. However, the parallel code is still there; if the routine exists outside a loop, its own loops will be executed in parallel.

- **-Miomutex** is only applicable to the Fortran compiler. Use this switch if your Fortran program performs I/O. This switch causes the compiler to place critical sections around the I/O statements of Fortran programs whether they are in loops or not. A critical section is a portion of the code that is only executed by one processor. If you are not trying to parallelize loops with I/O statements, specify **-Mnoiomutex**.

- **-Mreentrant** is only applicable to the Fortran compiler. That is, for the C compiler, **-Mreentrant** is the default, and **-Mnoreentrant** is ignored. **-Mreentrant** has two consequences for the Fortran compiler.

  1. The Fortran compiler does not perform any optimizations that might destroy reentrancy.

  2. The default variable allocation becomes stack-based. That is, it is as if the switch **-Mnosave** were issued. You can still specify particular variables as saved with SAVE declarations within your code.

  If you want to make calls within a parallelized loop, those calls must be compiled with -**Mreentrant**.

  When you specify -**Mreentrant**, you should be aware that your program's stack requirements may increase.

The **-Msave** switch has the opposite effect of -**Mreentrant**. Programs compiled with **-Msave** will have their local variables static, not stack-based. When compiling procedures intended to be thread-safe, do not use **-Msave** with **-Mreentrant**.

Compiler Directives

For best results, you should use compiler switches as well as directives or pragmas. If you use **-Mconcur** without directives or pragmas, the compiler will attempt to parallelize all loops that qualify, starting with the outermost loop. With directives or pragmas, you can target specific loops in your program for parallelization.

Directives or pragmas have three possible scopes: global, routine, and loop.

- Global means that the directive or pragma applies from its current position to the end of the file.

- Routine means that the directive or pragma applies from its current position to the end of the routine. This means that if you place a routine directive at the start of a subroutine, it applies to just that subroutine.

- Loop means that the directive or pragma applies to the next loop. Its effect is not passed through to any nested loops.

When you specify -**Mconcur**, the compiler attempts to parallelize all loops. With nested loops, the compiler starts at the outermost loop and does not continue deeper into the nesting once it successfully parallelizes. Most often you want to perform selective parallelization of loops. One approach is to turn off all parallelization with a directive and then specifically target certain loops for parallelization. Another approach is leave parallelization on and then specifically turn off parallelization for particular loops.

Fortran Directives

Fortran directives begin in the first column position with a *c*, just like a comment. The syntax is as follows:

cdir$*x value*

where *x* is one of **g** (for global), **r** (for routine), or **l** (for loop). *value* is the same keyword specified by the switch. For example, the *value* of the directive corresponding to the switch **-Mconcur** is either **concur** or **noconcur**; the *value* of the directive corresponding to the switch **-Mcncall** is either **cncall** or **nocncall**.

In the following example, the first directive turns off parallelization for the entire routine. The second directive tells the compiler to attempt to parallelize the outer loop indexed by *i*, but not the inner loop indexed by *j*.

```
cdir$r noconcur
cdir$l concur
       do 100 i=1,n
         do 100 j=1,m
                 .
                 .
                 .
100    continue
```

Fortran statements appear here

In the next example, the first directive turns off parallelization for the routine. The second directive tells the compiler to attempt to parallelize the inner loop indexed by *j*, but not the outer loop indexed by *i*.

```
cdir$r noconcur
       do 100 i=1,n
cdir$l concur
```

```
          do 100 j=1,m
                  .
                  .                            Fortran statements appear here
                  .
100    continue
```

C Pragmas

C pragmas have the following syntax:

# pragma *scope value*

where scope is one of **global**, **routine**, or **loop**. *value* is the same keyword specified by the switch. For example, the *value* of the directive corresponding to the switch **-Mconcur** is either **concur** or **noconcur**; the *value* of the directive corresponding to the switch **-Mcncall** is either **cncall** or **nocncall**.

In the following example, the first pragma turns off parallelization for the entire routine. The second pragma tells the compiler to attempt parallelize the outer loop indexed by *i*, but not the inner loop indexed by *j*.

```
# pragma routine noconcur
# pragma loop concur
   for(i=0; i<n; i++) {
      for(j=0; j<m; j++) {
              .
              .                            C statements appear here
              .
      }
   }
```

In the next example, the first directive turns off parallelization for the routine. The second directive tells the compiler to attempt to parallelize the inner loop indexed by *j*, but not the outer loop indexed by *i*.

```
# pragma routine noconcur
   for(i=0; i<n; i++) {
# pragma loop concur
      for(j=0; j<m; j++) {
              .
              .                            C statements appear here
              .
      }
   }
```

Getting Information

As you optimize your code, you may need more information about what the compiler has accepted for parallelization, what it has rejected, and why it rejected certain loops.

Setting **-Minfo=loop** causes the compiler to report information about both vectorization and parallelization. Previously, it reported only vectorization. By default, no information is displayed.

You may also choose to set **-Mneginfo=concur**. If the compiler rejects a loop for parallelization, this switch causes the compiler to print a message explaining why.

Additional Information about Loop Parallelization

This section describes some additional topics you should consider when doing SMP programming.

Reductions

A loop is non-parallelizable if it contains cross-iteration dependencies. That is, the result of one iteration is used in another iteration. A reduction is an operation that processes a vector of values and reduces them to a single scalar.

Technically, a reduction has a cross-iteration dependency; but parallelization is possible if the order of the reduction operations is altered. For example, summing the elements of a vector is a reduction operation.

```
          sum = 0
          do 100 i=1,n
             sum = sum + v(i)
    100   continue
```

Here, the value of sum from one iteration is used in the next iteration. But if each processor forms a partial sum, and then the two processors communicate and sum their partial sums, the same answer is obtained. This assumes, however, that the operations are commutative (their order does not matter). This is true mathematically, but there may be some subtle numeric effects that you need to take into consideration.

By default, the compiler will parallelize reduction loops. If you want to turn off the parallelization of reduction loops, use the **-Mconcur** switch and set it equal to **noassoc** as follows: **-Mconcur=noassoc**.

Note that you may have more than one assignment to **-Mconcur**. **-Mconcur** options are separated by commas. For example, you may want to turn off parallelization of reduction loops as well as set the distribution to cyclic. You would specify **-Moncur** as follows: **-Mconcur=dist:cyclic,noassoc**.

Namelist Groups

Namelist groups are used in Fortran programs. Local variables that appear in a namelist group are not placed on the stack, even if the module is compiled with **-Mreentrant**. They are statically allocated. If you compile with **-Mcncall** (which allows loops with procedures to be parallelized), then you should check that the variables in the namelist group do not introduce unwanted thread dependences. Treat the namelist variable as if it were declared in a SAVE statement.

Calls within Loops

It was stated earlier that **-Mconcur** linked in *libc_r.a*. This library is linked in for both C and Fortran programs. Fortran code requests *libf.a* and this is linked in automatically. *libf.a* uses either *libc.a* or *libc_r.a*; and if you specify the **-Mconcur** switch, *libc_r* is linked in.

libc_r.a contains reentrant versions of the C library calls. Note, however, that if you do not also specify **-Mcncall**, the compiler will not attempt to parallelize loops containing C library calls even though these C calls are thread-safe.

Basic Math Library Calls

When you compile your program with **-Mconcur**, the reentrant version of the Basic Math Library, *libkmath_r* is linked in. The Basic Math Library calls are then executed in parallel.

If you parallelize a loop that contains a Basic Math Library call, the call itself is not parallelized; but because the loop is parallelized, the call must be thread-safe.

Default Loop Thresholds

The compiler attempts to parallelize a loop only if its iteration count exceeds 100. If the loop is a reduction loop, its iteration count must exceed 200 before it is considered. Currently, these numbers are hard limits, and their values are subject to change.

Focus on Compute Node Processes

Best results are achieved if you limit loop parallelization to processes running on the compute node. Do not parallelize service node applications of the controlling process of a parallel application.

Writing Custom Pthread Applications

This section describes how to achieve SMP parallelization by writing your own pthread applications. If you write your own pthread applications, do not use any of the compiler switches discussed in "Compiler Switches" on page 6-5.

Limitations when Writing Custom Pthread Applications

- Currently, only the following libraries are thread-safe:

  libpthreads.a
  libc_r.a
  libm_r.a
  libkmath_r.a
  libksignal_r.a

  These libraries must be linked in explicitly if you are writing custom pthread applications.

- Any global variables used or set by a non-thread-safe library may also have to be protected. For example, if a non-thread-safe function sets the global variable *errno*, you must be sure to read the value of *errno* before allowing any other pthread to make any call that could change the value of *errno*. See "errno Confusion" on page 6-50 for more information about *errno*.

- On a GP node, all the pthreads in a process always run on the same processor. Scheduling of pthreads is handled by the kernel, which uses a policy of time sharing with aging. You cannot control or get information about pthread scheduling by using pthread library calls. On an MP node all pthreads run on both available application processors.

- Pthreads use kernel resources as well as user-level resources (to be specific, each pthread uses one *kernel thread*). This means that using very large numbers of pthreads can exhaust certain resources within the kernel.

- The *POSIX Threads Extension [C language] P1003.4a/D4 (Draft 4), August 1990* includes an optional feature called "thread priority scheduling." This feature is not available in the current release. If you attempt to make use of this feature, you will get compilation errors (for use of an unsupported data type), link errors (for use of an unsupported library call), or run-time errors (for use of an unsupported system call). If a run-time error occurs, the call fails with the *errno* value **ENOSYS**.

- There is no Fortran interface to the pthreads package. If you must use pthreads in a Fortran program, you could make the calls to the pthreads library from a C function, which can then be compiled to a *.o* file and linked into the Fortran program. However, this programming model has not been tested.

- The application development tools currently are not thread-aware and do not have any features to support pthreads.

  In particular, IPD does not support the debugging of threaded applications. Similarly, the debugging of core files generated by threaded applications is not supported.

- Pthread-specific data are bound to a pthread key with **pthread_setspecific()**. The key is created with **pthread_keycreate()**. Note that all pthread-specific data for a process reside in one memory page, which on Paragon systems is 4K bytes. Hence, you may have **pthread_keycreate**() fail on ENDMEM when your system still has memory available.

Recommended Safe Operating Environment

The previous section described the limitations which *cannot* be exceeded. This section recommends limitations which you *should not* exceed in the current release. Exceeding these limitations may result in unexpected behavior, up to and including system crashes and data loss.

- No process should have more than six pthreads at once. This limitation is due to the availability of system resources and the use of the reentrant C library.

  Any pthread can create or terminate another pthread at any time. The system does not impose a limit on the number of active pthreads in a process, on a node, or in the whole system. However, the total active pthreads per process (including the main thread) should be kept at or below six. Exceeding this limit may result in an emulator exception.

  Under some circumstances, you may wish to risk creating more than six pthreads. If you do so, any additional pthreads should only perform computation and not make calls to *libc_r.a.* The stack size is one of the system resources that limits the number of pthreads, and so decreasing the stack size may allow you to create more pthreads. Creating more than six pthreads, however, remains currently unsupported.

- Only one pthread in a process should use the message-passing calls described in Chapter 3. The message-passing pthread can be the main thread or another pthread, but a pthread other than the main thread will experience higher message latency than the main thread.

  This limitation is due to the fact that the message-passing library (*libnx.a*) is not thread-safe. Also, there is no mechanism in current message passing calls to send or receive messages to or from a specific pthread within a process. See "Message Passing and Pthreads Library Calls" on page 6-46 for more information.

  If more than one pthread in a process attempts to perform message passing, message-passing performance may degrade, incorrect information may be returned from an **info...**() call, and global operations such as **gsync**() may give unexpected results.

- All global operations (such as **gsync()**) must be performed by the message-passing pthread. This is necessary because all global operations use message-passing to synchronize the nodes. You can synchronize pthreads *within a process* by using a global variable counter as a barrier.

- Only one pthread in a process should use the parallel file I/O calls described in Chapter 5. The I/O pthread can be the main thread or another pthread. See "File I/O and Pthreads Library Calls" on page 6-47 for more information.

- The calls **gopen()** and **setiomode()** use message-passing internally. If the I/O pthread is not also the message-passing pthread, you must make sure that these calls are not used at the same time as any message-passing calls in the message-passing pthread.

- The standard OSF/1 file I/O system calls, such as **read()** and **write()**, can be called from multiple pthreads at the same time *if* they are called from a controlling process and they are only used with files that reside in UFS file systems. Otherwise, only one pthread in a process can use them.

- Applications with multiple pthreads should not be run in gang-scheduled partitions. Gang scheduling of threaded applications has not been thoroughly tested.

- Do not call **sigwait()** to wait on synchronous signals (those that are generated synchronously as the results of a pthread's faults, such as **SIGBUS** and **SIGSEGV**). Doing this may cause the application or the system to crash. See "Managing Signals" on page 6-44 for more information about **sigwait()**.

- Do not use calls from *libpthreads.a* or *libc_r.a* within a signal handler. Some of these calls use mutexes internally, which may result in deadlock (the handler can find itself waiting on an unavailable mutex lock, while the mutex lock cannot be released until the signal handler has returned).

- Asynchronous cancellation is very destructive and should be avoided. In particular, attempting to cancel a pthread doing file I/O on a PFS or UFS file system can cause the entire application to hang. See "Canceling Pthreads" on page 6-38 for more information about asynchronous cancellation.

- Do not terminate a process when other pthreads are progressing. For example, calling **exit()** or returning from **main()** kills all threads and terminates the entire process. If there are any other pthreads in the process, including pthreads generated transparently by library calls, null processes may result. Be sure to terminate all pthreads gracefully before terminating the program. In particular, be sure that all asynchronous and interrupt-driven message and I/O operations (such as **hrecv()** or **iread()**) are complete before the program terminates. It is also a good idea to ensure that **main()** always exits by calling **pthread_exit()**, never by calling **exit()** or by reaching the closing brace of **main()**. The last active pthread should call **exit()** to terminate process execution.

- No Mach calls can be used in a pthreads program. The Mach kernel interface (*libmach.a*) is not supported in the current release; use of Mach features in pthreads programs can cause pthreads internal errors or system crashes.

Compiling and Linking a Pthread Application

When compiling a program that uses the pthreads package, you should define the symbol **_REENTRANT**; this symbol ensures that thread-safe definitions are used in all included header files. (The compiler switch **-M**[**no**]**reentrant** does *not* have any effect on whether or not the resulting code is thread-safe. It only determines whether or not the code can be called recursively.)

When linking a program that uses the pthreads package, you must link in the library *libpthreads.a*, followed by the library *libc_r.a*. If you use the **-nx** switch, it can appear on the command line either before these two libraries or after them; if you use the **-lnx** switch, it should appear after both these libraries. The standard C library is also linked by default, but only after searching all libraries specified on the command line.

For example:

- To compile and link a non-parallel program:

```
% cc -D_REENTRANT -o node node.c -lpthreads -lc_r
```

- To compile and link a controlling process:

```
% cc -D_REENTRANT -o node node.c -lpthreads -lc_r -lnx
```

- To compile and link a parallel application:

```
% cc -D_REENTRANT -o node node.c -lpthreads -lc_r -nx
```

- To compile and link a parallel application with the reentrant math library:

```
% cc -D_REENTRANT -o node node.c -lpthreads -lc_r -lm_r -nx
```

Using Reentrant C Library Calls

The reentrant C calls are used by both Fortran and C programs. Although you cannot invoke these calls directly from a Fortran program, certain Fortran routines call them internally.

Only the calls in the reentrant C library (*libc_r.a*), the calls in the pthreads library (*libpthreads.a*), the calls in the math libraries (*libm_r.a* and *libkmath_r.a*), and the calls in the optional signal processing library (*libksignal_r.a*) are guaranteed to be thread-safe. Any calls to other libraries must be protected so that no two pthreads can call them at the same time. Table 6-1 lists the calls in *libc_r.a*.

Table 6-1. Calls in Reentrant C Library (*libc_r.a*) (1 of 2)

abort()	**endgrent()**	**fstatfs()**	**getsockname()**	**memcpy()**
abs()	**endpwent()**	**fsync()**	**getsockopt()**	**memmove()**
accept()	**endttyent()**	**ftell()**	**gettimeofday()**	**memset()**
access()	**endusershell()**	**ftruncate()**	**gettimer()**	**mkdir()**
acct()	**endutent()**	**ftw()**	**getttyent_r()***	**mknod()**
adjtime()	**exec_with_loader()**	**funlockfile()***	**getttynam_r()***	**mkstemp()**
advance()	**execlp()**	**fwrite()**	**getuid()**	**mktemp()**
alarm()	**execvp()**	**gcvt()**	**getusershell_r()***	**mktime()**
alloca()	**exit()**	**getaddressconf()**	**getutent_r()***	**mktimer()**
asctime_r()*	**fabs()**	**getc()**	**getutid_r()***	**mmap()**
async_daemon()	**fchdir()**	**getchar()**	**getutline_r()***	**modf()**
atexit()	**fchmod()**	**getclock()**	**getw()**	**mount()**
atof()	**fchown()**	**getcwd()**	**getwc()**	**mprotect()**
atoi()	**fclose()**	**getdirentries()**	**getwd()**	**msem_init()**
atol()	**fcntl()**	**getdtablesize()**	**gmtime()**	**msem_lock()**
bcmp()	**fcvt_r()***	**getegid()**	**gmtime_r()***	**msem_remove()**
bcopy()	**fdopen()**	**getenv()**	**htonl()**	**msem_unlock()**
bind()	**feof()**	**geteuid()**	**htons()**	**msgctl()**
brk()	**ferror()**	**getfh()**	**initstate()**	**msgget()**
bzero()	**fflush()**	**getfsent_r()***	**initstate_r()***	**msgrcv()**
calloc()	**ffs()**	**getfsfile_r()***	**insque()**	**msgsnd()**
catclose()	**fgetc()**	**getfsspec_r()***	**ioctl()**	**msync()**
catgets()	**fgets()**	**getfsstat()**	**isalnum()**	**munmap()**
catopen()	**fileno()**	**getgid()**	**isatty()**	**mvalid()**
chdir()	**flock()**	**getgrent_r()***	**isdigit()**	**nfssvc()**
chmod()	**flockfile()***	**getgrgid_r()***	**isnan()**	**nice()**
chown()	**fopen()**	**getgrnam_r()***	**isnand()**	**nl_langinfo()**
chroot()	**fpathconf()**	**getgroups()**	**isnanf()**	**ntohl()**
clearerr()	**fpgetmask()**	**gethostid()**	**isspace()**	**ntohs()**
clock()	**fpgetround()**	**gethostname()**	**isupper()**	**open()**
close()	**fpgetsticky()**	**getitimer()**	**isxdigit()**	**opendir()**
closedir()	**fprintf()**	**getlogin_r()***	**kill()**	**pathconf()**
connect()	**fpsetmask()**	**getpagesize()**	**ldexp()**	**perror()**
creat()	**fpsetround()**	**getpeername()**	**link()**	**pipe()**
ctermid()	**fpsetsticky()**	**getpgrp()**	**listen()**	**plock()**
ctime_r()*	**fputc()**	**getpid()**	**localtime_r()***	**poll()**
cuserid()	**fputs()**	**getppid()**	**longjmp()**	**printf()**
dbm_close()	**fread()**	**getpriority()**	**lseek()**	**profil()**
dbm_fetch()	**free()**	**getpwent_r()***	**lstat()**	**ptrace()**
dbm_open()	**freopen()**	**getpwnam_r()***	**madvise()**	**putc()**
dup()	**frexp()**	**getpwuid_r()***	**malloc()**	**putchar()**
dup2()	**fscanf()**	**getrlimit()**	**memccpy()**	**puts()**
ecvt_r()*	**fseek()**	**getrusage()**	**memchr()**	**pututline_r()***
endfsent()	**fstat()**	**gets()**	**memcmp()**	**putw()**

* Does not exist in the standard C library (*libc.a*).

Table 6-1. Calls in Reentrant C Library (*libc_r.a*) (2 of 2)

putwc()
quotactl()
raise()
rand()
rand_r()*
random()
random_r()*
re_comp_r()*
re_exec_r()*
read()
readdir()
readdir_r()*
readlink()
readv()
realloc()
reboot()
recv()
recvfrom()
recvmsg()
reltimer()
remque()
rename()
revoke()
rewind()
rewinddir()
rindex()
rmdir()
rmknod()
rmtimer()
sbrk()
scanf()
select()
semctl()
semget()
semop()
send()
sendmsg()
sendto()
setbuf()
setbuffer()
setclock()
setfsent()
setgid()
setgrent()
setgroups()
sethostid()
sethostname()
setitimer()
setjmp()
setlinebuf()
setlocale()
setlogin()
setpgid()
setpgrp()
setpriority()
setpwent()
setregid()
setreuid()
setrlimit()
setsid()
setsockopt()
setstate()
setstate_r()*
settimeofday()
setttyent()
setuid()
setusershell()
setutent()
setvbuf()
shmat()
shmctl()
shmdt()
shmget()
shutdown()
sigaction()
sigaddset()
sigdelset()
sigemptyset()
sigfillset()
sigismember()
signal()
sigprocmask()
sigreturn()
sigstack()
sigsuspend()
sleep()
socket()
socketpair()
sprintf()
srand()
srandom()
srandom_r()*
sscanf()
stat()
statfs()
strcat()
strchr()
strcmp()
strcpy()
strcspn()
strdup()
strerror_r()*
strftime()
string()
strlen()
strncat()
strncmp()
strncpy()
strpbrk()
strrchr()
strspn()
strtok_r()*
strtol()
strtoul()
swapon()
symlink()
sync()
sysconf()
table()
tempnam()
time()
times()
tmpfile()
tmpnam()
tolower()
truncate()
ttyname_r()*
tzset()
ulimit()
umask()
umount()
uname()
ungetc()
unlink()
unlocked_fclose()*
unlocked_fflush()*
unlocked_fread()*
unlocked_fseek()*
unlocked_fwrite()*
unlocked_getc()*
unlocked_getchar()*
unlocked_getwc()*
unlocked_putc()*
unlocked_putchar()*
unlocked_setvbuf()*
utimes()
utmpname()
vfork()
vfprintf()
vprintf()
vsprintf()
wait4()
waitpid()
write()
writev()

* Does not exist in the standard C library (*libc.a*).

The calls in *libc_r.a* can be divided into three groups according to their names:

- Most of the calls in *libc_r.a* have the same names as calls in the standard C library (*libc.a*). These calls generally work the same as the equivalent calls in *libc.a*. However, they perform special checks and locks internally to be sure they will work if called by multiple pthreads at the same time. Also, many blocking system calls only block the calling pthread instead of every pthread of the process. The following commonly-used calls have the following effects in programs with multiple pthreads:

exit() Kills all pthreads of the process and closes all opened files. See "Calling exit()" on page 6-51 for more information on using **exit()** in programs with multiple threads.

fork() Copies only the calling pthread to the new process's address space. If a mutex lock is held by another pthread, then the calling pthread in the new process may deadlock. For example, if a process has two pthreads and pthread 0 calls **fork()** when pthread 1 is holding a mutex lock inside a call to **printf()**, only the pthread in the new process will hang when it calls **printf()**.

exec() Kills all pthreads other than the calling pthread, then loads a new program into the process's address space. The result is a new program with one pthread. The calling pthread can create additional pthreads if it wants.

chdir() Changes the current working directory for all pthreads in the calling process.

sleep() Puts only the calling pthread to sleep.

wait() Blocks only the calling pthread; does not return until every pthread of the process being waited for exits. Note that a pthread can **wait()** for a process created by a different pthread.

perror() Uses the per-pthread *errno* (see "errno Confusion" on page 6-50).

Note that these calls may have different semantics on different platforms:

- *libc_r.a* also includes some calls whose names end in **_r**. These calls do the same thing as the similarly-named calls in the standard C library, but have different parameters.

  - If the **_r** call is the only version of the call in *libc_r.a*, you must use the **_r** call to be sure your code is thread-safe. This is the case for most of the **_r** calls.

  - If both an **_r** and a non-**_r** version of the call exist in *libc_r.a*, both versions are thread-safe, but the **_r** version offers better performance. This is the case for **gmtime()**, **initstate()**, **rand()**, **random()**, **readdir()**, **setstate()**, and **srandom()**.

  The **_r** calls are noted with an asterisk in Table 6-1.

- Finally, the calls **flockfile()**, **funlockfile()**, and **unlocked_...()** exist only in *libc_r.a*:

  flockfile() Locks the specified standard I/O stream for exclusive use by the calling pthread.

  funlockfile() Unlocks the specified stream.

unlocked_...() Performs operations on a stream while it is locked (these are called "unlocked" calls because they do not perform any locking or unlocking of their own).

The **unlocked_...()** calls are not thread-safe by themselves; they must be used together with **flockfile()/funlockfile()**.

These calls offer better I/O performance and more control over I/O from pthreads than the standard thread-safe I/O calls. For example, the thread-safe version of **putc()** locks out all other I/O calls, writes the specified character, then unlocks. If you write a series of characters to a file with **putc()**, this locking and unlocking results in considerable overhead; also, there is nothing to prevent characters written by two different pthreads from becoming intermingled.

You can instead use **flockfile()** to lock out all other operations on the file, a series of **unlocked_putc()** calls to write characters without locking and unlocking, and finally a **funlockfile()** to release the lock. In this case only one pair of lock/unlock operations is performed; your I/O performance will be better, and no other pthread's output can interfere. See the *OSF/1 Programmer's Reference* for more information about these calls.

Using Pthreads Library Calls

This section tells you how to use the calls in *libpthreads.a* to create and control pthreads in your programs. See the *OSF/1 Programmer's Reference* for more detailed information on each call.

Pthreads Library Data Types and Symbols

In order to use any calls from the pthreads library, your program must include the file *<pthread.h>*, which defines several types and symbols used by this library. The most important of these are:

pthread_t Within each process, each active pthread is identified by a unique *pthread ID*, which is a value of type *pthread_t*. You use a pthread's ID to identify the pthread in all calls that control pthreads.

pthread_mutex_t and *pthread_cond_t*

Each active mutex is identified by a value of type *pthread_mutex_t*, and each active condition variable is identified by a value of type *pthread_cond_t*.

pthread_attr_t, *pthread_mutexattr_t*, and *pthread_condattr_t*

Objects of these types, called *attributes objects*, are used to specify the attributes (characteristics) of pthreads, mutexes, and condition variables. These types are extensible, and can support new features added by later revisions of the pthreads standard while maintaining compatibility with existing programs. Objects of these types are created with default values and can be changed by pthreads library calls.

pthread_attr_default, *pthread_mutexattr_default*, and *pthread_condattr_default*
These are external symbols whose values are the default attributes for an object of the appropriate type. If you want to create an object with the default attributes, you can use one of these symbols instead of creating a new attributes object with the default attributes.

pthread_key_t This data type supports the per-pthread global data structure in the pthreads library. This enables different functions to access global data that only belongs to a single pthread.

The Main Thread

Each program initially has a single thread—the flow of control that starts at the beginning of the function **main**(). This thread is referred to as the *main thread.*

Any other pthreads in the program are created by the main thread, either directly or indirectly. But threads do not have a parent-child relationship, as processes do, so the main thread does not have any special relationship with or control over other pthreads in the process.

However, the C library treats the function **main**() specially, in a way that can affect other threads in the process:

NOTE

If the function **main()** returns (either by executing a **return** statement or by reaching the closing brace of the function), the C library generates an implicit call to **_exit()**, which kills all pthreads in the process and terminates the process.

This means that you must either have **main**() wait for all other pthreads before returning, or make sure that **main**() always terminates itself by calling **pthread_exit**() rather than calling **exit**() or returning.

Managing Pthread Execution

Synopsis	Description
int **pthread_create**(pthread_t **thread*, pthread_attr_t *attr*, void *(**routine*)(void **arg*), void **arg*);	Creates a pthread.
pthread_t **pthread_self**(void);	Returns the ID of the calling pthread.
int **pthread_equal**(pthread_t *thread1*, pthread_t *thread2*);	Compares two pthread identifiers.
void **pthread_yield**(void);	Allows the scheduler to run another pthread instead of the current one.
void **pthread_exit**(void **status*);	Terminates the calling pthread.
int **pthread_join**(pthread_t *thread*, void ***status*);	Waits for a pthread to terminate.
int **pthread_detach**(pthread_t **thread*);	Detaches a pthread. Invalidates the specified pthread ID.

To create a pthread, call **pthread_create**(). This call has the following parameters:

thread Pointer to a variable of type *pthread_t* that receives the pthread ID of the newly-created pthread.

attr An object of type *pthread_attr_t* that describes the desired attributes of the new pthread. This can be the default pthread attribute object *pthread_attr_default*, or a user-created pthread attribute object (see "Managing Pthread Attributes" on page 6-22).

routine Pointer to the initial function to be executed by the new pthread. This function is assumed to return *void* * and to have one argument of type *void* *.

arg Value of type *void* * to be passed to the initial function as its argument.

A pthread can determine its own pthread ID by calling **pthread_self**(), and a pthread ID can be compared against another pthread ID by calling **pthread_equal**(). Note that *pthread_t* is an "opaque" type, and you should not use standard C operators on it.

The **pthread_yield()** call will decrease the priority of the calling pthread and give up the node's processor to other pthreads that have higher priorities than the calling pthread. The kernel decides which thread to run next, based on its time sharing and aging policies. Eventually, the calling pthread will be scheduled to run again when other pthreads become lower priority pthreads. A pthread should call **pthread_yield()** to give up the processor when it is making no progress or has no work to do.

A pthread terminates when it calls **pthread_exit()** or returns from its initial function. However, the termination of a pthread does not release all the resources associated with the pthread. To release a terminated pthread's resources, a different pthread must call **pthread_join()** or **pthread_detach()**:

- **pthread_join()** blocks until the specified pthread terminates, then releases the specified pthread's resources and returns the exit status of the specified pthread to its caller. The exit status is the value specified in the pthread's **pthread_exit()** call, or the return value of its initial function if it did not call **pthread_exit()**.

- **pthread_detach()** tells the pthreads library to release the specified pthread's resources, then returns immediately to its caller. Later, when the specified pthread terminates, the library releases the pthread's resources and discards the pthread's exit status. Once a pthread has been detached, any subsequent calls to **pthread_join()** or **pthread_cancel()** specifying that pthread will fail.

Any pthread that creates other pthreads should call **pthread_join()** or **pthread_detach()** for each pthread it created before it terminates itself.

Managing Pthread Attributes

Synopsis	Description
int **pthread_attr_create**(pthread_attr_t **attr*);	Creates a pthread attributes object.
int **pthread_attr_setstacksize**(pthread_attr_t **attr*, long *stacksize*);	Sets the value of the stack size attribute of a pthread attributes object.
int **pthread_attr_delete**(pthread_attr_t **attr*);	Deletes a pthread attributes object.
int **pthread_attr_getstacksize**(pthread_attr_t *attr*);	Returns the value of the stack size attribute of a pthread attributes object.

The only pthread attribute that is currently modifiable is stack size. (A pthread's priority and scheduling policy are managed by the kernel and cannot be inspected or changed.) To set a pthread's stack size, use the following procedure:

1. Call **pthread_attr_create()** to create a pthread attributes object (an object of type *pthread_attr_t*).

2. Call **pthread_attr_setstacksize()** to set the stack size in that object.

3. Use the modified pthread attributes object in the call to **pthread_create()** that creates the pthread.

4. Call **pthread_attr_delete()** to remove the pthread attributes object.

Once a pthread has been created, the size of its stack is fixed and can't be changed.

To use the default stack size, you can simply use the default pthread attribute object *pthread_attr_default* instead of creating your own pthread attributes object.

You can use **pthread_attr_getstacksize()** to find out the current stack size in a pthread attributes object.

NOTE

Whenever possible, use the same stack size for all pthreads. Be sure to check the stack size. The default stack size increased for Release 1.3.

Each pthread is built on a lower-level construct called a *kernel thread.* When you create a pthread, the pthread library tries to re-use a kernel thread from a pool of existing kernel threads. This means that creating a new pthread is more expensive if there are no existing kernel threads inside the pthreads library that can be reused. A major cause for the kernel being unable to recycle kernel threads is using a different stack size for new pthreads; this should be avoided.

Managing Mutexes

Synopsis	Description
int **pthread_mutex_init**(pthread_mutex_t *mutex, pthread_mutexattr_t attr);	Creates a mutex.
int **pthread_mutex_lock**(pthread_mutex_t *mutex);	Locks a mutex.
int **pthread_mutex_trylock**(pthread_mutex_t *mutex);	Tries once to lock a mutex.
int **pthread_mutex_unlock**(pthread_mutex_t *mutex);	Unlocks a mutex.
int **pthread_mutex_destroy**(pthread_mutex_t *mutex);	Deletes a mutex.

A pthread *mutex* is a binary semaphore with two states: *locked* and *unlocked.* When a mutex is created, its initial state is unlocked. Only one pthread at a time can lock a mutex. When a pthread successfully locks a mutex, it becomes the mutex's *owner.* Any other pthread that attempts to lock the mutex will block until the owner unlocks the mutex. Mutexes cannot be used recursively: if the owner attempts to lock the mutex again, the attempt fails.

You should use mutex locks to serialize pthread access to a block of code that accesses a nonshareable resource, such as a file or a non-thread-safe library. A pthread that is waiting on a mutex lock will not use any of the node's processor time.

To create and initialize a mutex, call **pthread_mutex_init()**. This call creates a new mutex with the attributes specified by *attr* (typically the default mutex attributes object *pthread_mutexattr_default*) and stores the new mutex's ID into the variable pointed to by *mutex.* A newly-created mutex is unlocked.

To lock a mutex, call **pthread_mutex_lock()**. The call to **pthread_mutex_lock()** will block the calling pthread until the mutex lock is available. A pthread waiting on a mutex lock will be scheduled out and another pthread will be scheduled to run. When the calling pthread is again scheduled to run because no higher-priority pthread can run, it checks the availability of the mutex lock again and is scheduled out again if the mutex lock is still unavailable.

Note that there is no guarantee that a pthread waiting on a **pthread_mutex_lock()** will eventually get the lock. If you do not want to block until the lock is available, call **pthread_mutex_trylock()**. This call tries once to lock the specified mutex. If the attempt succeeds, the call returns 1 immediately; but if the mutex is already locked, the call returns 0 immediately.

When a pthread is finished using the resource controlled by the mutex, it should release the lock by calling **pthread_mutex_unlock()**. This allows any other pthread that has been waiting to lock the mutex to proceed.

When all pthreads have finished using the mutex, you should remove it and release all resources associated with it by calling **pthread_mutex_destroy()**. You cannot destroy a mutex that is currently locked. Attempting to lock or unlock a mutex that has been successfully destroyed will result in undefined behavior.

Managing Mutex Attributes

Synopsis	Description
int **pthread_mutexattr_create**(pthread_mutexattr_t **attr*);	Creates a mutex attributes object.
int **pthread_mutexattr_delete**(pthread_mutexattr_t **attr*);	Deletes a mutex attributes object.

No mutex attributes are currently defined. You can either use the default mutex attributes object, *pthread_mutexattr_default*, or create a mutex attribute object for use in **pthread_mutex_init()** by calling **pthread_mutexattr_create()**. A user-created mutex attributes object should be released by calling **pthread_mutexattr_delete()**.

An Example Pthreads Program

The following program demonstrates some principles of using pthreads and mutexes. It creates a user-specified number of pthreads, each of which prints its node number, ptype, and pthread ID and the message "Done."

```
#include <pthread.h>
#include <stdlib.h>
#include <nx.h>

#define MAXTHREAD       6                       /* thread maximum limit */

/* pthread resources */
pthread_t               thread[MAXTHREAD];      /* per-thread pthread ID */
pthread_mutex_t         mutex;                  /* mutex to protect global
                                                   variable "thread_alive" */

/* global variables that only the main thread writes to */
int     max_thread = MAXTHREAD;                 /* maximum thread number */
int     my_node;                                /* my node number */
int     my_ptype;                               /* my ptype */

/* shared global variable that is modified by all threads */
int     thread_alive;                           /* count of living threads */

/* forward declarations */
void thread_fun(int thread_id);                 /* initial function for
                                                   new threads */

main(int argc, char *argv[])
{
    int     index;                          /* loop index */
    int     my_thread = 0;                  /* main thread is indexed 0 */

    my_node = mynode();
    my_ptype = myptype();

    if(argc != 2) {
        if(my_node == 0) {
            printf("Usage: %s <nthreads>\n", argv[0]);
        }
        exit(1);
    }

    max_thread = atoi(argv[1]);

    if(max_thread > MAXTHREAD) {
        if(my_node == 0) {
            printf("Error: %d threads requested, must be %d or less\n",
                   max_thread, MAXTHREAD);
        }
        exit(1);
    }

    /* The main thread is the last thread alive, so don't count itself. */
    thread_alive = max_thread - 1;

    /* create and initialize a mutex to control access to "thread_alive" */
    if(pthread_mutex_init(&mutex, pthread_mutexattr_default) == -1) {
        perror("pthread_mutex_init Error");
    }

    /*
    * Spawn threads and remember each thread's pthread ID.
    * The main thread is indexed as thread 0.
    */
    thread[my_thread] = pthread_self();
    for(index = 1; index < max_thread; index++) {
        if(pthread_create(&thread[index], pthread_attr_default,
                          (void *)thread_fun, (void *)index) == -1) {
            perror("pthread_create Error");
            exit(2);
        }
    }

    /* loop until all other threads are finished. */
    while(thread_alive != 0) {
        pthread_yield();
    }

    /*
    * Ignore other threads' exit status (can also be done right after
    * pthread_create()).
    */
    for(index = 1; index < max_thread; index++) {
        pthread_detach(&thread[index]);
    }

    printf("( %3d, %3d, %3d) Done\n", my_node, my_ptype, my_thread);
}


/*************************************************************************
* thread_fun() -- This is the initial function for new threads
*************************************************************************/
```

```
void thread_fun(int my_thread)
{

    printf("( %3d, %3d, %3d) Done\n", my_node, my_ptype, my_thread);

    /*
    * use mutex to protect global variable "thread_alive"
    */
    if(pthread_mutex_lock(&mutex) == -1) {
        perror("pthread_mutex_lock Error");
    }
    thread_alive--;
    if(pthread_mutex_unlock(&mutex) == -1) {
        perror("pthread_mutex_unlock Error");
    }

    /* terminate (status is ignored) */
    pthread_exit(NULL);
}
```

Assuming this program is called *pthreads.c*, use the following command to compile it:

```
% cc -D_REENTRANT -o pthreads pthreads.c -lpthreads -lc_r -nx
```

To run this program with three pthreads per process on two nodes of your default partition, use the following command:

```
% pthreads 3 -sz 2
```

The results of this application run:

```
(   0,   0,   2) Done
(   1,   0,   1) Done
(   0,   0,   1) Done
(   0,   0,   0) Done
(   1,   0,   2) Done
(   1,   0,   0) Done
```

Note that the results may appear in a different order on each run.

Using Condition Variables to Synchronize Pthreads

Synopsis	Description
int **pthread_cond_init**(pthread_cond_t **cond*, pthread_condattr_t *attr*);	Creates a condition variable.
int **pthread_cond_wait**(pthread_cond_t **cond*, pthread_mutex_t **mutex*);	Waits on a condition variable.
int **pthread_cond_timedwait**(pthread_cond_t **cond*, pthread_mutex_t **mutex*, struct timespec **abstime*);	Waits on a condition variable for a specified period of time.
int **pthread_cond_signal**(pthread_cond_t **cond*);	Wakes up a pthread that is waiting on a condition variable.
int **pthread_cond_broadcast**(pthread_cond_t **cond*);	Wakes up all pthreads that are waiting on a condition variable.
int **pthread_cond_destroy**(pthread_cond_t **cond*);	Destroys a condition variable.

A condition variable is a construct that implements synchronization among pthreads. There are other ways of implementing synchronization, but for many applications condition variables are the most straightforward.

For example, consider two pthreads. Assume that one is a producer and the other a consumer. The producer could be a pthread that received messages, and the consumer could be a pthread that processed the messages. The producer places the messages in a buffer shared with the consumer. The consumer must have some way of knowing when the buffer contains messages to process.

You could program the consumer to continually check a flag that's set by the producer. If the consumer does this, it is spinning, which is wasteful of CPU cycles. Alternatively, you could program the consumer to periodically check a flag. This is polling, which, although not as wasteful as spinning, still is profligate of computer cycles.

Better yet, the consumer could block until the producer signals it that messages are ready for processing. The pthreads package provides *condition variables* for this purpose.

Condition variables use the following objects:

- A *mutex* (an object of type *pthread_mutex_t*, as discussed under "Managing Mutexes" on page 6-24) that is used to protect the pthread condition variable and the global predicate variable.

- A *condition variable* (an object of type *pthread_cond_t*) that links all pthreads waiting on a particular condition.
- A *global predicate variable* that indicates the current state of the condition. It could be a global integer variable.

To create and initialize a condition variable, call **pthread_cond_init()**. This call creates a new condition variable with the attributes specified by *attr* (typically the default condition attributes object *pthread_condattr_default*) and stores the new condition variable's ID into the variable pointed to by *cond*. The list of pthreads that are waiting on the new condition variable is initially empty.

To wait for a condition, call **pthread_cond_wait()**. This call unlocks the specified mutex and blocks until the specified condition is signaled by another pthread. When the condition is signaled, the call re-locks the mutex and returns to the caller. **pthread_cond_timedwait()** is similar, but if the specified amount of time passes before the condition is signaled, the call re-locks the mutex and returns an error condition. You must successfully lock the specified mutex before calling **pthread_cond_wait()** or **pthread_cond_timedwait()**, and you should unlock the mutex after the call to **pthread_cond_wait()** or **pthread_cond_timedwait()** returns.

To signal a condition, call **pthread_cond_signal()** or **pthread_cond_broadcast()**. **pthread_cond_signal()** signals the specified condition to one of the pthreads that is waiting for it (if more than one pthread is waiting for the condition to be signaled, the kernel selects one of them arbitrarily). **pthread_cond_broadcast()** signals the specified condition to all of the pthreads that are waiting for it. If no other pthread is waiting on the condition, these calls have no effect. No mutex lock is required for these calls (but a mutex can be used to prevent certain race conditions; see the example on page 6-34).

If a pthread calls **pthread_cond_wait()** *after* the specified condition has been signaled, that pthread could wait forever. To prevent this problem, use a *global predicate variable*. This can be any variable that is visible to all pthreads. You use it as follows:

- Before calling **pthread_cond_signal()** or **pthread_cond_broadcast()**, a pthread should set the value of the condition's global predicate value to indicate that the condition has occurred. The global predicate value should be protected by a mutex if there is any possibility that more than one pthread could try to set it at once.
- Before calling **pthread_cond_wait()** or **pthread_cond_timedwait()**, a pthread should check the current value of the condition's global predicate variable. If the condition has already occurred, the pthread should proceed without calling **pthread_cond_wait()** or **pthread_cond_timedwait()**.
- After a successful call to **pthread_cond_wait()** or **pthread_cond_timedwait()**, a pthread should check the global predicate value to be sure it has the expected value. If the global predicate variable does not indicate that the condition has occurred, the pthread should call **pthread_cond_wait()** or **pthread_cond_timedwait()** again.

The first example shown under "Examples of Condition Variables" on page 6-33 gives an example of this technique.

When all pthreads have finished using a condition variable, you should remove it and release all resources associated with it by calling **pthread_cond_destroy()**. You cannot destroy a condition variable that is currently being waited on.

Managing Condition Attributes

Synopsis	Description
int **pthread_condattr_create**(pthread_condattr_t **attr*);	Creates a condition variable attributes object.
int **pthread_condattr_delete**(pthread_condattr_t **attr*);	Deletes a condition variable attributes object.

No condition attributes are currently defined. You can either use the default condition attributes object, *pthread_condattr_default*, or create a condition attribute object for use in **pthread_cond_init**() by calling **pthread_condattr_create**(). A user-created condition attributes object should be released by calling **pthread_condattr_delete**().

Examples of Condition Variables

The following example uses a mutex to protect the global predicate variable *cond_true*, which is used to prevent the signaling pthread from calling **pthread_cond_signal()** until the waiting pthread has called **pthread_cond_wait()**. Note that the call to **pthread_cond_signal()** is within the mutex lock; this is not necessary, but can prevent certain race conditions.

The pthread waiting for the condition executes the following code:

```
if(pthread_mutex_lock(&mutex) == -1) {
    perror("pthread_mutex_lock Error");
}

/*
* If the expected condition already exists, don't call
* pthread_cond_wait(), since the condition signal will not be
* sent if no threads are waiting for this condition.
*
* Recheck the state of cond_true after calling
* pthread_cond_wait() to ensure that the received condition
* signal is for this expected state change.
*/
while(!cond_true) {
    /*
    * mutex will be unlocked in pthread_cond_wait() when
    * calling thread is ready to wait.
    */
    if(pthread_cond_wait(&cond, &mutex) == -1) {
        perror("pthread_cond_wait Error");
        break;
    }
    /*
    * mutex will be locked again when the calling thread
    * is awakened.
    */
}

if(pthread_mutex_unlock(&mutex) == -1) {
    perror("pthread_mutex_unlock Error");
}
```

The pthread signaling the condition executes the following code:

```
if(pthread_mutex_lock(&mutex) == -1) {
    perror("pthread_mutex_lock Error");
}

/* This global variable needs a mutex's protection. */
++cond_true;

/*
* The pthread_cond_signal() call does not use a mutex internally.
* The mutex protection will guarantee that every thread can
* catch the expected condition signal once it calls
* pthread_cond_wait().  This will prevent the endless block of
* the calling thread.
*/
if(pthread_cond_signal(&cond) == -1) {
    perror("pthread_cond_signal Error");
}

if(pthread_mutex_unlock(&mutex) == -1) {
    perror("pthread_mutex_unlock Error");
}
```

Here's another example, which uses **pthread_cond_broadcast()** to wake up all waiting pthreads.

```
/*
* Simulate a thread-level gsync() to synchronize all active
* threads at a barrier.  A counter in this example can only be
* used once.
*/

long              cond_gsync;  /* counter of threads arrived */
long              max_thread;  /* number of active threads */
pthread_mutex_t   mutex;
pthread_cond_t    cond;
```

```
void thread_gsync(long *cond_gsync)
{
    if(pthread_mutex_lock(&mutex) == -1) {
        perror("pthread_mutex_lock Error");
    }

    /* Increase the count of threads that have called
       thread_gsync() */
    *cond_gsync++;

    if(*cond_gsync == max_thread) {
        /*
        * If I'm the last thread to call thread_gsync(),
        * wake up all threads waiting on this condition.
        */
        if(pthread_cond_broadcast(&cond) == -1) {
            perror("pthread_cond_broadcast Error");
        }
    } else {
        while(*cond_gsync != max_thread) {
            /*
            * Other threads haven't called thread_gsync() yet,
            * so wait for them in pthread_cond_wait().
            */
            if(pthread_cond_wait(&cond, &mutex) == -1) {
                perror("pthread_cond_wait Error");
                break;
            }
        }
    }

    if(pthread_mutex_unlock(&mutex) == -1) {
        perror("pthread_mutex_unlock Error");
    }
}


main()
{
    /*
    * Initialize counter and the condition that counter
    * will meet.
    */
    max_thread = atoi(argv[1]);  /* threads to create */
    cond_gsync = 0;              /* init counter */

    if(pthread_cond_init(&cond,
                         pthread_condattr_default) == -1) {
        perror("pthread_cond_init Error");
    }

    if(pthread_mutex_init(&mutex,
                          pthread_mutexattr_default) == -1) {
        perror("pthread_mutex_init Error");
    }
        •
        •
        •
    /* Create more pthreads */
        •
        •
        •
}

    /* Every thread calls thread_gsync() once */
    thread_gsync(&cond_gsync);
    /* Every thread passes this barrier at the same time */
```

Here's an example using **pthread_cond_timedwait()**:

```
#include <sys/timers.h>

long    interval = 10;          /* 10 seconds interval */

struct timespec abs_time;

/* Get the current time */
getclock(TIMEOFDAY, &abs_time);

/*
* Can use another member of structure timespec to specify the
* waiting interval in nanoseconds. But the resolution cannot be
* smaller than the interval between updates of the system clock.
*
* The wait time should not be so small that the
* absolute time specified is smaller than the
* time spent inside the pthread_cond_timedwait() call.
*/
abs_time.tv_sec = abs_time.tv_sec + interval;

if(pthread_mutex_lock(&mutex) == -1) {
    perror("pthread_mutex_lock Error");
}

while(!condition) {
    if(pthread_cond_timedwait(&cond, &mutex, &abs_time) == -1) {
        /* EAGAIN is the timeout error code. */
        if(errno != EAGAIN) {
            perror("pthread_cond_timedwait Error");
        }
        break;
    }
}

if(pthread_mutex_unlock(&mutex) == -1) {
    perror("pthread_mutex_unlock Error");
}
```

Canceling Pthreads

Synopsis	Description
int **pthread_cancel**(pthread_t *thread*);	Requests cancellation of a pthread.
int **pthread_setcancel**(int *state*);	Enables or disables the general cancelability of the calling pthread.
int **pthread_setasynccancel**(int *state*);	Enables or disables the asynchronous cancelability of the calling pthread.
void **pthread_testcancel**(void);	Creates a cancellation point in the calling pthread.

The pthreads package includes a *pthread cancellation mechanism* that allows a pthread to terminate the execution of other pthreads. The call **pthread_cancel()** requests cancellation of the specified pthread; however, the specified pthread may terminate later or not at all, depending on its *cancelability states.*

Cancelability States

Each pthread has two *cancelability states* that determine how it reacts to cancellation requests. Each of the two states can be set to the value **CANCEL_ON** (enabled) or **CANCEL_OFF** (disabled).

- If *general cancelability* determines whether or not the pthread can be canceled:

  - If general cancelability is enabled, cancellation requests are accepted. Cancellation may or may not occur immediately, depending on the *asynchronous cancelability* state.

  - If general cancelability is disabled, cancellation requests are queued until general cancelability is enabled again.

  General cancelability is enabled by default; a pthread can change its general cancelability state by calling **pthread_setcancel()**.

- When general cancelability is enabled, a second cancelability state called *asynchronous cancelability* determines how quickly the cancellation occurs:

  - If asynchronous cancelability is enabled, when a cancellation request is received the pthread begins termination immediately.

- If asynchronous cancelability is disabled, when a cancellation request is received the pthread does not begin termination until it reaches a *cancellation point.* The default cancellation points are calls to **pthread_cond_wait()**, **pthread_cond_timedwait()**, **pthread_join()**, and **pthread_setcancel(CANCEL_ON)**. A pthread can also create an explicit cancellation point by calling **pthread_testcancel()**, which otherwise does nothing.

Asynchronous cancelability is disabled by default; a pthread can change its asynchronous cancelability state by calling **pthread_setasynccancel()**.

NOTE

Asynchronous cancelability should not be enabled in the current release.

Asynchronous cancellation of certain pthreads, particularly pthreads performing file I/O, can cause the entire application to hang.

NOTE

You must be careful not to cancel a pthread that is holding a mutex lock.

Canceling a pthread that is holding a mutex lock leaves the mutex locked with no way to unlock it, possibly resulting in deadlock. For example, a pthread calling **printf()** will get a mutex lock inside the reentrant C library. A cancellation of this pthread during the call to **printf()** will cause all other pthreads calling **printf()** to deadlock.

Functions such as **printf()**, which can cause deadlock if they are canceled, are called *not safe to cancel.*

NOTE

Most library functions are not safe to cancel.

In particular, all of the calls in *libnx.a* are not safe to cancel. The list of functions that *is* safe to cancel can be found in the **pthread_setasynccancel()** manpage in the *OSF/1 Programmer's Reference.*

Cancellation Examples

Here's an example of changing a pthread's cancelability states:

```
/* flip the general cancelability of the calling thread */
if(pthread_setcancel(CANCEL_ON) == -1) {
    perror("pthread_setcancel Error");
}

if(pthread_setcancel(CANCEL_OFF) == -1) {
    perror("pthread_setcancel Error");
}


/* flip the asynchronous cancelability of the calling thread */
if(pthread_setasynccancel(CANCEL_ON) == -1) {
    perror("pthread_setasynccancel Error");
}

if(pthread_setasynccancel(CANCEL_OFF) == -1) {
    perror("pthread_setasynccancel Error");
}
```

Here's an example of delivering and accepting cancellations:

```
pthread_t   thread_id;  /* value from pthread_create() call */
    •
    •
    •
/*
* Cancel another thread whose pthread ID is "thread_id".
*/
if(pthread_cancel(thread_id) == -1) {
    perror("pthread_cancel Error\n");
}
    •
    •
    •
/*
* If a cancellation request is already posted, this call will
* not return.
*/
pthread_testcancel();
/* Execution continues if no posted cancellation request */
```

Pthreads Cleanup Routines

Synopsis	Description
void **pthread_cleanup_pop(** int *execute*);	Removes a routine from the top of the cleanup stack of the calling pthread and optionally executes it.
void **pthread_cleanup_push(** void (**routine*)(void **arg*), void **arg*);	Pushes a routine onto the cleanup stack of the calling pthread.

Pthreads may have resources that must be released before the pthread terminates. Each pthread can create a list of cleanup routines, called the *cleanup stack*, to release those resources. The routines on the cleanup stack are called, in order from top to bottom, when the pthread terminates for any of the following reasons:

- Calling **pthread_exit()**.
- Returning from its initial function.
- Being cancelled by another pthread.

To place a function on the cleanup stack, call **pthread_cleanup_push()**; to remove the top function from the cleanup stack, call **pthread_cleanup_pop()**. You can optionally execute the function as it is popped. Every call to **pthread_cleanup_push()** must be matched with a **pthread_cleanup_pop()** call in the same *lexical scope* (that is, within the same set of "{ ... }" braces).

If general cancelability is enabled, whenever a pthread allocates a resource it should push a function that deallocates that resource onto the cleanup stack; when the pthread is finished with the resource it should deallocate it by popping the function off the cleanup stack and executing it. This ensures that all resources are accounted for if the pthread is cancelled.

Managing Pthread Keys

Synopsis	Description
int **pthread_keycreate**(pthread_key_t **key*, void (**destructor*)(void **value*));	Creates a key to be used with pthread-specific data.
int **pthread_setspecific**(pthread_key_t *key*, void **value*);	Binds a pthread-specific value to a key.
int **pthread_getspecific**(pthread_key_t *key*, void ***value*);	Returns the value bound to a key.

The pthreads package provides *pthread-specific data objects* to associate information with individual pthreads. Each pthread-specific data object is controlled by a *key* (an object of type *pthread_key_t*). A pthread creates a new key by calling **pthread_keycreate()**, associates the key with a pthread-specific data object by calling **pthread_setspecific()**, and then retrieves the data associated with the key by calling **pthread_getspecific()**. See the *OSF/1 Programmer's Reference* for more information on these calls.

Executing a Routine Once

Synopsis	Description
int **pthread_once**(pthread_once_t **once_block*, void(**routine*)());	Calls an initialization routine.

The **pthread_once()** call executes the specified routine the first time it is called (from any pthread), and does nothing every subsequent time. The parameter *once_block* must be declared as **static**. For example:

```
static pthread_once_t  init_once;

void lib_util_init() {
    /* perform some initialization that can only be done once */
}
    •
    •
    •
/*
* Every pthread calls pthread_once(), but only the first one
* executes lib_util_init().
*/
if(pthread_once(&init_once, lib_util_init) == -1) {
    perror("pthread_once Error");
}
```

Managing Signals

Synopsis	Description
int **sigwait**(sigset_t *set);	Suspends the calling pthread until one of a specified set of signals is received.

The **sigwait()** call is used to turn asynchronous signals into synchronous notifications. Before calling **sigwait()**, you must create a *signal set*, using the standard signal calls **sigemptyset()**, **sigfillset()**, **sigaddset()**, and **sigdelset()**, and then block the signals in that set from being delivered. When you call **sigwait()** with that signal set, the calling pthread is suspended until one or more of the signals in the set is received by the process containing the pthread. If one of the specified signals was received (and blocked) before the call to **sigwait()**, the call returns immediately. **sigwait()** returns the signal number of the signal that was received.

The **sigwait()** call only works for *asynchronous* signals (those that are generated externally from the pthread, such as those generated by **kill()** in other processes or by the user pressing **`<Ctrl-\>`**). Contrast this with **sigaction()**, which only works for *synchronous* signals (those that are generated as the result of the pthread's faults, such as **SIGBUS**). If both **sigaction()** and **sigwait()** are used on the same signal, the results are unspecified.

NOTE

In a parallel application, sending an asynchronous signal to an application's controlling process affects the controlling process (as specified by the controlling process's signal mask), and also causes the signal to be broadcast to the compute processes. In an application linked with **-nx**, the controlling process's signal mask is always the default.

See "Signals and Pthreads Library Calls" on page 6-48 for more information on signals in applications with multiple pthreads.

Here's an example that uses **sigwait()** to deal with the asynchronous signal **SIGQUIT**. This example uses a parent process to generate the asynchronous signal by calling **kill()**.

```
long    sig;
long    ret;

sig = SIGQUIT;

pid = fork();
```

```
if(pid == -1) {
    perror("fork()");
    exit(1);
} else if(pid != 0) {
    /* parent process */
    sleep(2);
    /*
    * Deliver the signal SIGQUIT to child process.
    */
    if(kill(pid, sig) == -1) {
        perror("kill ");
        exit(1);
    }
    exit(0);
}

/* child process */
if(sigemptyset(&set) != 0) {
    perror("sigemptyset");
}

/* Add the signal SIGQUIT to the signal mask */
if(sigaddset(&set, sig) != 0) {
    perror("sigaddset");
}

/* Block the signal SIGQUIT from delivery */
if(sigprocmask(SIG_BLOCK, &set, NULL) != 0) {
    perror("sigprocmask()");
}

/*
* During the next 10 seconds, the posted signal from the
* parent process becomes a pending signal.
*/
sleep(10);

/*
* sigwait() blocks the calling thread until the specified signal
* arrives, then unblocks and returns the value SIGQUIT.
*/
if((ret = sigwait(&set)) == -1) {
    perror("sigwait()");
} else {
    printf("Received signal %d, expected %d\n", ret, sig);
}

/*
```

```
* The thread can decide what to do with this signal.
*/

/*
* There is no destructive default action of core dump on the
* posted signal SIGQUIT.
*/
```

Interfacing with Non-Thread-Safe Code

Whenever you call a non-thread-safe library from a process with multiple pthreads, you must make sure that no two pthreads call the same library at the same time. There are two ways do this:

- Make sure that only one pthread ever calls the library.
- Use mutexes to protect all calls to the library.

Here's an example of the second technique:

```
pthread_mutex_lock(&mutex);
non_thread_safe_call();
pthread_mutex_unlock(&mutex);
```

Note that the *same* mutex must be used by all pthreads for any calls from the same library. If all calls to the non-thread-safe library are surrounded by a lock and unlock of the same mutex, as shown here, any pthread that calls the library while another pthread is currently calling it will block until the other pthread returns and unlocks the mutex. See "Managing Mutexes" on page 6-24 for more information on mutexes.

Message Passing and Pthreads Library Calls

Message-passing is done on a process-by-process basis. All message-sending calls specify the recipient by node and process type; there is no way to specify a particular pthread within that process (all the threads in a process have the same process type). Similarly, when a message arrives at a process, there is nothing to prevent confusion among pthreads; for example, a pthread could probe for a message, find a pending message of the specified type, and then attempt to receive it—only to find that another pthread has already received it. For this reason, you should make sure that only one pthread in each process uses message-passing calls.

You should also keep the following special considerations in mind when using message-passing and pthreads in the same application:

- Blocking calls, such as **csend()**, only block the calling pthread, not the entire process. While the calling pthread is blocked, other pthreads can continue to run. The pthread that is blocked releases processor resources until the **csend()** returns.

- When the message-passing pthread uses one of the global calls (those described under "Global Operations" on page 3-27), the call blocks until the message-passing pthread on every other node makes the same call. If one of those message-passing pthreads is blocked (for example, by a mutex lock), the operation will hang all the message-passing pthreads in the application.
- An **hsend()/hrecv()** handler can use calls from *libpthreads.a* or *libc_r.a* (note that *libc_r.a* includes almost the entire C library). However, you should avoid calls from *libpthreads.a*. In particular, do not call **pthread_exit()** because the **hsend()/hrecv()** handler is implemented as a pthread.
- An **hsend()/hrecv()** handler also should not use the **info...()** calls. Because the handler executes concurrently with the main message-passing pthread, the **info...()** calls may return values representing messages received by the main message-passing pthread. The main message-passing pthread can use **masktrap()** to protect critical regions from the handler.
- If an **hsend()/hrecv()** handler performs any message passing, you must put **masktrap()** calls around any message-passing calls in the main message-passing pthread that could be called while the handler is active. Otherwise, any **info...()** calls in the handler could reflect the value of a message received by the main message-passing pthread.

  In addition, any **info...()** call in the main program must be within the same set of **masktrap()** calls as the message-receiving call to which it applies. Otherwise, the **info...()** call in the main message-passing pthread could reflect the value of a message received by the handler.

File I/O and Pthreads Library Calls

In general, opened files are per-process resources. A pthread can open a file, a second pthread can use the open file descriptor to write or read, and a third pthread can use the same file descriptor in an **lseek()** call. The movement of file pointers is visible to all pthreads, so if multiple pthreads are accessing the same file they must coordinate their actions with mutexes or condition variables. However, blocking calls such as **read()** and **cwrite()** only block the calling pthread, not the entire process.

If two pthreads doing file I/O read and write concurrently, they can each read and write their own data independently. If you are performing I/O to a file in a synchronized PFS I/O mode (see "Using I/O Modes" on page 5-14), the synchronization information is stored with the file descriptor; each file is synchronized independently.

See "Recommended Safe Operating Environment" on page 6-12 for limitations on using I/O calls from multiple pthreads.

nx_nfork() and nx_initve...() and Pthreads Library Calls

In a controlling process with multiple pthreads:

nx_nfork() Copies only the calling pthread to the new process on each node. Can only be called from one pthread.

nx_initve...() If the user's shell is the Bourne shell (**sh**), the **nx_initve...()** calls perform a **fork()** internally. As described under "Using Reentrant C Library Calls" on page 6-14, **fork()** copies only the calling pthread to the new process. This means that if there are multiple pthreads in the calling process before the call to **nx_initve...()**, after the **nx_initve...()** all pthreads except the calling pthread will appear to cease to exist. (If the user's shell is **ksh** or **csh**, this problem does not exist.)

nx_initve...() can be called at most once in a process. This means that at most one pthread in a process can call it.

Signals and Pthreads Library Calls

The following special considerations apply to signals in programs with multiple pthreads.

Signal Types

There are two types of signals: synchronous and asynchronous:

- *Synchronous* signals are caused by a pthread's own actions, such as when a pthread divides by zero (causing a **SIGFPE** signal) or attempts to access memory outside its address space (causing a **SIGSEGV** signal).

- *Asynchronous* signals are caused by something external to the pthread, such as another process calling **kill()** or the user pressing `<Ctrl-\>` on the keyboard (causing a **SIGQUIT** signal).

If a pthread causes a synchronous signal, the handler routine executes in the context of that pthread only. If an asynchronous signal is delivered to a process, the handler routine executes in the context of the main thread.

Signals are a Per-Process Resource

Signals are generally managed as per-process objects in pthreads programs. Signal masks, signal handlers, and signal sending and receiving are all oriented toward the process, not toward a particular pthread. This means that signals affect the entire process. Of particular interest:

- A **SIGSTOP** signal stops all pthreads of the receiving process.

- A **SIGCONT** signal continues all pthreads of the receiving process.

- If one pthread of a program with multiple pthreads causes a **SIGSEGV** or **SIGBUS**, the entire process (not just the faulting pthread) receives the signal. If this signal has not been handled, all pthreads are killed and the program core dumps.

It is important to be aware that a pthread program's signal mask has a per-process visibility. In other words, all pthreads share the same mask. If one pthread changes its mask (for example, by calling **sigprocmask()**) the change affects all pthreads. The thread-safe **sigwait()** requires manipulation of the signal mask, as does **sigaction()** and other common signal-management routines.

Along with the signal mask, signal handlers are also process-wide objects. A signal handler can be registered for the process (for example, by calling **sigaction()** or **sigwait()**) by any pthread. Because the handlers are process-wide objects, a second pthread registering a handler for a given signal will override the handler registered by the first pthread.

In general, blocking calls only block the calling pthread. This is the case with **sigsuspend()** as it is with **wait()**, **sleep()** etc. Note also, that if multiple pthreads are blocking on **sigsuspend()** for a given signal, all pthreads will continue when that signal arrives. This differs from **sigwait()** which unblocks only one of the pthreads.

sigwait() creates a hidden pthread which manipulates the process's signal mask and registers a signal handler for each signal **sigwait()** has been asked to wait for. Because of this, use of other signal management calls (especially **sigaction()**) on the signals being waited for, would be hazardous. Care must be taken when changing a signal mask so the state of a **sigwait()**ed signal's bit is not changed.

The floating pont exeception mask operates differently. It has a per-pthread visibility, not a per-process visibility. **fpsetmask()** only affects the calling pthread's floating point exception mask. The exception mask is not inherited by any created pthread; each pthread must set its own floating point exception mask.

Dealing with Signals

A way to deal with signals in a pthread application is the following:

- Use **sigaction()** to catch the synchronous signals. **sigaction()** only works with synchronous signals.

- Use **sigprocmask()** to block the asynchronous signals, then **sigwait()** to receive the signal as a notification. **sigwait()** only works with asynchronous signals.

Do not use **sigwait()** and **sigaction()** on the same signal.

Handling Errors

The handling of error situations in a program with multiple pthreads should be robust and graceful. It should protect the pthreads that did not cause the error from being interrupted or terminated. It also should give information on which pthread caused the error and coordinate a proper shutdown of all pthreads if the error is fatal. If the error cannot be recovered from by the pthread causing the error, and other pthreads are depending on this pthread to progress, it might be best to terminate those pthreads right away and shut down the rest of the pthreads later.

errno Confusion

The *errno* variable is set to an error value when a system or library call fails. An immediate call to **perror()** or **nx_perror()** reads the value of *errno* and prints out the error message corresponding to its current value. User-written code may also set *errno* to take advantage of this standard error-handling mechanism.

In multiple-threaded programs, there are two *errno* variables: a global (per-process) *errno* variable and a local (per-pthread) *errno* variable. The preprocessor symbol **_REENTRANT** determines (for some libraries) which of these *errno* variables the symbol *errno* refers to. If **_REENTRANT** is defined at the point the file *<errno.h>* is included, the symbol *errno* refers to the per-pthread *errno*. Otherwise, the symbol *errno* refers to the global (per-process) *errno*.

In the current release, the libraries provided with the operating system are not consistent in their use of the two different *errno* variables.

- *libpthreads.a*, *libm_r.a*, and *libkmath.a* set and reference only the per-pthread *errno*.
- *libc_r.a* sets both the global and the per-pthread *errno*, but only references the per-pthread *errno*.
- All other libraries set and reference only the global *errno*.

The inconsistency in the way libraries treat *errno* is what causes the confusion. For example, if a call to *libnx.a* fails in a program, the global *errno* is set. But if the program is compiled with **_REENTRANT**, the calling pthread can only see its local *errno* whose value does not reflect the error in the *libnx.a* call that just failed.

In general, this means that code compiled with **_REENTRANT** cannot use *errno* values returned by non-thread-safe libraries. However, because some non-thread-safe libraries make calls to the standard C library, some *errno* values *are* usable. For example, **cread()** (in the non-thread-safe library *libnx.a*) calls **read()**. If you link with *libc_r.a*, any error that occurs in **read()** will be reflected in the per-pthread *errno* and will be visible to the calling pthread. But any error that occurs in the **cread()** call itself (before or after the call to **read()**) will *not* be visible to the calling pthread.

Rather than access *errno* values directly, you can use the **perror()** and **nx_perror()** calls. These calls print on standard error a specified argument string and a short message that depends on the value of *errno.*

- The **perror()** call in *libc_r.a* uses the per-pthread *errno.*
- The **nx_perror()** call in *libnx.a* uses the global *errno*. Because *libnx.a* is not thread-safe, results from **nx_perror()** may be suspect. Note that calls in *libnx.a* may use calls in *libc.a*. Specifically, **nx_perror()** uses the **perror()** in *libc.a*. The **perror()** call in *libc.a* uses the global *errno.*

Calling exit()

Calling **exit()** when an error occurs terminates the entire process and closes any opened files. For this reason, it's a bad idea to call **exit()** on an error returned from a system call inside a pthread. Instead, you should call **pthread_exit()** to terminate the failing pthread and return a value indicating failure to the pthread that calls **pthread_join()**. The pthread that calls **pthread_join()** should use this information to shut down all other pthreads properly.

Use of Underscore Versions of Paragon System Calls

The standard versions of most Paragon system calls in *libnx.a* terminate the calling process when an error occurs and send a message to standard error describing the error. For example, **isend()** will call **nx_perror()** to print out the error message, then call **exit()** to terminate the process. This implies that if a pthread causes an error in an **isend()** call, this error will kill the rest of the pthreads in the process.

For this reason, in programs with multiple pthreads you should always use the underscore versions of these calls instead. For example, calling **_isend()** will return -1 and set the global *errno* in case of error, instead of terminating the process. The calling pthread can then use this information to shut down the rest of the pthreads cleanly. See "Handling Errors" on page 4-55 for more information on underscore calls.

Catch Signals Causing Core Dump by Default

The default action for the signals **SIGFPE** (floating point exception) and **SIGSEGV** (segmentation violation) is to core dump, terminate the process, and terminate the application. This will also kill all pthreads in the application.

When this occurs, you want to be able to figure out which pthread was responsible for this problem. For synchronous signals, the best way to do this is to install a signal handler to catch them and print out the pthread ID when the signal is received. For asynchronous signals, use **sigwait()** to catch them and then terminate all pthreads gracefully.

Avoid Core Dumps

If your application has a number of pthreads, very large core files may be generated. It could take several minutes to dump one of these large core files. Creating core dumps in a program with many pthreads is not supported.

When One Pthread Hangs

When debugging a program with multiple pthreads, always keep track of every active pthread, as much as possible, to detect the hang of a single pthread. Knowing which pthread has hung will help you determine the cause of a program hang.

Designing a Parallel Application 7

Introduction

This chapter describes some general design guidelines to follow when writing parallel applications. However, the best way to become skilled in parallel programming is to do it. With that in mind, this chapter presents three examples of parallel applications. Each example is intended to illustrate a different aspect of parallel design technique.

- The first example is a nearly-perfectly-parallel application that evaluates a definite integral to calculate π. This example illustrates how a sequential application can be ported to a parallel system with minimal effort. Much of the sequential algorithm can be maintained. The parallel design consists of separating the user interface from the core computation and then distributing that core computation onto the nodes.

- The next example is the multiplication of a matrix by a vector. In addition to the numerical technique, this example illustrates the use of parallel file I/O by assuming a matrix that is too large to reside entirely in memory.

- The third example solves a classic computer science problem called the N-Queens problem. Given a chess board with N x N grid locations, where can you place N queens so that no queen is under attack? This example illustrates a technique called control decomposition. This technique also appears in more complicated real-world applications such as electronic design rule checking.

Programming Model

As described in Chapter 1, the Paragon supercomputer is a distributed-memory parallel computer with a high-speed interconnect network. The following characteristics of the system should be kept in mind when designing or porting applications:

- The system is made up of an ensemble of processor/memory pairs called *nodes*. The nodes do not share memory. They present a single system image (for example, a process running on one node can send a signal to a process running on another node), but the nodes operate independently of each other.

- All the nodes are fully connected. They communicate with each other and the host by passing messages.

- Each node executes its own program. In many applications, it turns out that each node executes the same program on a different set of input data. There may be some conditional code that identifies one or more nodes that perform special actions.

These characteristics influence the design of parallel applications, as described in the remainder of this chapter.

Parallel Programming Techniques

Parallel applications have varying degrees of parallelism. A *perfectly-parallel* application is one that requires no internode communication. In a perfectly-parallel application, if you double the number of nodes, you halve the computation time.

Most applications involve a mix of computation and internode communication; in these applications, increasing the number of nodes reduces the computation time, but can never yield a "perfect" speedup. The more time a program spends communicating instead of computing, the less speedup you get by adding nodes.

In order to get the best possible speed from a parallel program, you must design it so that each node spends as much time as possible computing, and as little time as possible communicating (or waiting for communication). Here are some techniques that can help you to do this:

- Separate the user interface from the computational parts of the code.

- Distribute the computation among the nodes so that their computational load is evenly balanced.

- Write your application so that you can run it on more nodes, thus improving performance, without having to recode.

- Design your internode communication such that the nodes spend as little time in communication (or waiting for communication) as possible.

The following sections tell you more about these techniques.

Separating the User Interface from the Computation

To have each node do as much computation, and as little non-computational work, as possible, you should analyze the algorithm and separate the user interface from the computational kernel. You can designate one of the nodes to handle the user interface, or put the user interface in the application's *controlling process* (see "The Controlling Process" on page 4-26 for information on this process). In either case, the part of the program that handles the user interface and the part of the program that does the computation communicate by passing messages.

In the π example, node 0 requests the number of integration intervals from the user. It then sends that number to the other nodes, and all the nodes do the calculation.

Balancing the Load

You should keep all the nodes busy and have them finish at the same time, because if some nodes have to wait for others to finish, they're wasting cycles doing nothing. Analyze your application and distribute the computation among the nodes so that their computational load is evenly balanced.

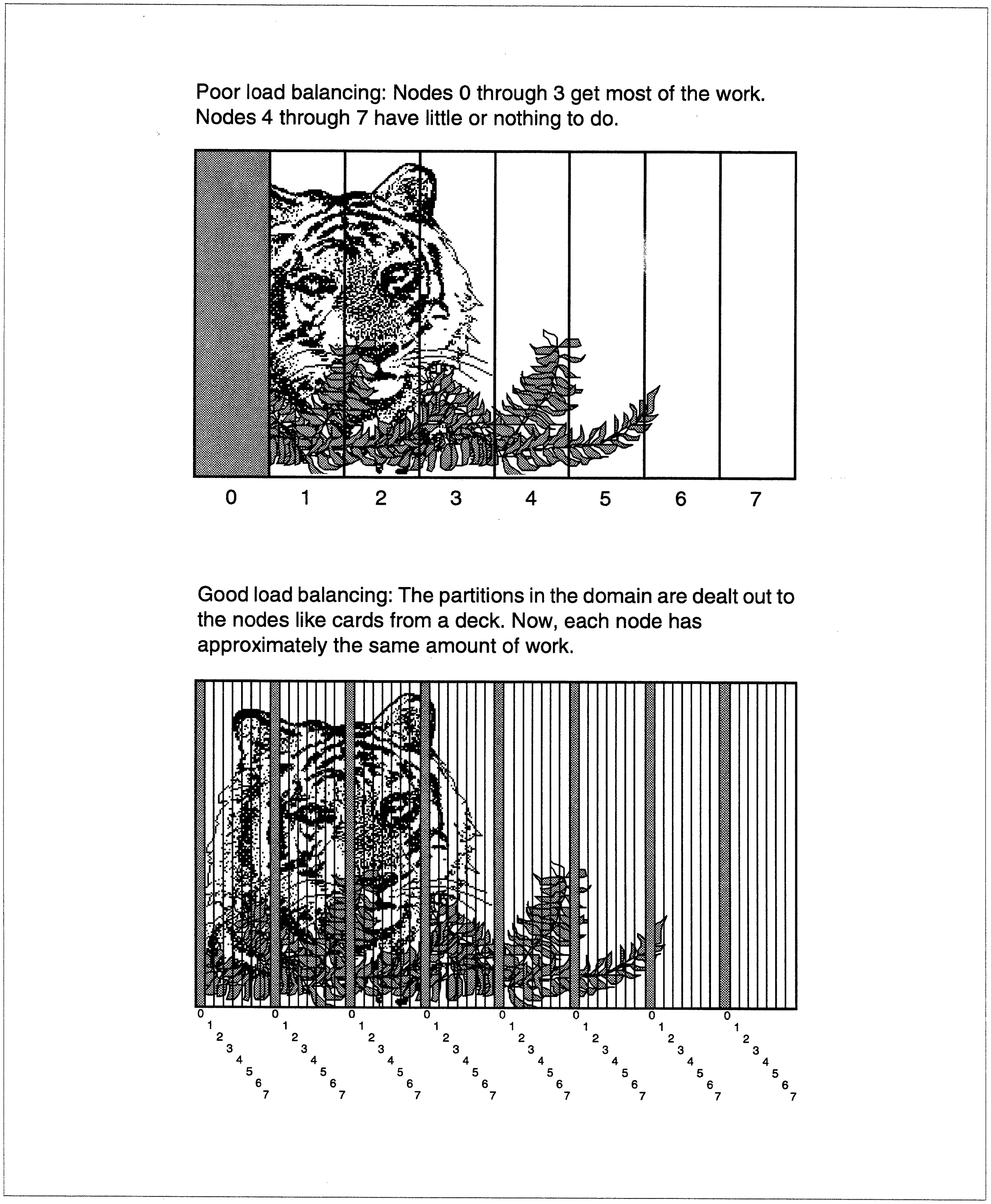
The process of distributing a problem among the nodes is referred to as *problem decomposition*, or just *decomposition.* There are two kinds of decomposition: *domain decomposition* and *control decomposition.*

Domain Decomposition

In domain decomposition, the input data (the domain) is partitioned and assigned to different processors. How you divide and distribute the data among the nodes can have a significant effect on the efficiency of your application.

For example, consider an application that performs image enhancement (see Figure 7-1). Because some parts of the image may be more detailed than others, they will require more processing. The shaded portion of Figure 7-1 shows the work done by node 0. If you divide the image sequentially among the nodes, as shown in the top half of Figure 7-1, some nodes may get a partition that requires a lot of work and other nodes may get a partition that requires little or no work. In the top half of Figure 7-1, node 0 gets a lot of work and node 7 gets no work at all. This is inefficient.

You can often achieve better load balancing by dividing the image into smaller partitions and then distributing the partitions sequentially among the nodes, as shown in the bottom half of Figure 7-1. This is analogous to dealing out the partitions like cards in a deck; it spreads out the work more evenly among the nodes. As the bottom half of Figure 7-1 shows, each node gets some slices that require a lot of work, some slices that require a moderate amount of work, and some slices that require no work. This is more balanced and efficient for this type of problem, and may be appropriate for your problem as well.



Poor load balancing: Nodes 0 through 3 get most of the work. Nodes 4 through 7 have little or nothing to do.

Good load balancing: The partitions in the domain are dealt out to the nodes like cards from a deck. Now, each node has approximately the same amount of work.

Figure 7-1. Using Domain Decomposition to Achieve Load Balancing

Control Decomposition

Control decomposition, on the other hand, divides the *tasks* to be performed rather than the *data*. For many problems, this is a more natural decomposition.

For example, consider a tree-search used in a game-playing algorithm. Assume that you're at some mid-level of the tree. You could approach the problem as a domain decomposition and divide the current branches among the nodes. Each node would then follow its branch down to the leaves and then return the leaves as an answer. The leaves in this case are the possible moves. Depending on the current state of the game, some of the branches may be quite involved and require a great deal of processing. Other branches may be simple. The result is that some nodes finish before others. This is a poor problem decomposition.

Approaching this problem as a control decomposition achieves better load balancing. In a control decomposition, you think of the branches not as data partitions but rather as tasks that need to be performed.

To manage these tasks, you have to introduce a little bureaucracy. Assign one node as a manager node. This manager node then gives tasks to idle nodes. When the node finishes a task, it reports its answer and requests another task. It's this "reporting for duty" that characterizes a control decomposition.

The manager node must, of course, do some initial setup. For example, it may follow the tree down until the number of branches exceeds the number of available nodes by some predetermined factor.

This method produces the best results when the tasks assigned near the end of the problem are about the same size. For example, if one of the last tasks assigned was a very long task, the other nodes may be idle while that last node finishes.

The N-Queens example (presented later in this chapter) shows control decomposition.

Making the Program Independent of the Number of Nodes

You should write your application so that you can run it on more nodes, thus improving performance, without having to recode.

This method also turns out to be the most natural one to use when porting an existing sequential application. After you've separated the user interface from the core computation, you still have a sequential algorithm, but you can think of it as the special case of an application that runs on one node. Once you have done this, you can parallelize the computation part for an arbitrary number of nodes.

The π example illustrates this technique. The number of nodes appears only as the variable *nodes*.

Designing Your Communication Strategy

Your should design your internode communication such that the nodes spend as little time in communication as possible. This may involve running some tests to determine an optimal message length. Often, you can decrease the number of messages by increasing the size of each message. You may also be able to improve communication performance by using asynchronous message-passing calls, as described under "Asynchronous Send and Receive" on page 3-10.

Using Global Operations

You should use the global operations, described under "Global Operations" on page 3-27, when possible. That section described a simple example of a global sum. Using **gdsum()** results in a significant improvement over having one node perform the global sum by explicitly collecting all the partial sums. Also, after the execution of the **gdsum()**, the global sum is available on each node.

The matrix*vector example in this chapter uses another global operation called **gcolx()**. In that example, a large vector is distributed over the nodes. **gcolx()** collects the components from each node and constructs the complete vector on each node. As with **gdsum()**, the answer is available on each node.

Using Alternate Node Topologies

The nodes in the Paragon supercomputer are connected in either a hypercube or a mesh network. However, because of the specialized message-passing hardware in both architectures, communication with distant nodes is nearly as fast as communication with neighboring nodes. This means that you do not have to structure your application's communications as a hypercube or mesh; you can choose an alternate topology that makes more sense for your program. This can make your program easier to write and understand, at a tiny cost in performance.

When you use an alternate node topology, you embed your node topology (a *virtual topology*) into the nodes' actual network topology (the *physical topology*). One example of a virtual topology is the ring. This topology is useful in certain types of many-body calculations. The technique consists of partitioning the particles into groups and assigning each group to a different node. A node then calculates the state of its group. This state information is then passed to another node which calculates the state of its own particles and takes into account the state received from the previous node. The state information moves from node to node around a ring. You can implement a ring topology by writing a function like this one:

```
succ(int n)
{
    int maxnode;
    maxnode = numnodes() - 1;

    if ( (n >= 0) && (n < maxnode))
        return(n+1);
    else if (n == maxnode)
        return(0);
    else
        return(-1);
}
```

Given a valid node ID (*n*), this function returns the node ID of the successor of node *n* in a ring embedded in a partition with **numnodes()** nodes. Else it returns -1. (The predecessor function is similar.) A node can send a message to process type 0 on its successor node with the following **csend()** call:

```
csend(MSGTYPE, buf, sizeof(buf), succ(mynode()), 0);
```

Example Application: Calculating pi

This application uses an n-point quadrature rule to evaluate the following definite integral:

$$\pi = \int_0^1 \frac{4}{(1 + x^2)} dx$$

Admittedly, using the power of a Paragon supercomputer for such a simple application is overkill, but the application demonstrates concepts that are just as valid for more challenging problems.

Here is a sequential program (written in Fortran) that evaluates the above integral. The source for this program is available on the Paragon supercomputer in */usr/share/examples/fortran/pi/piserial.f.* Note that the user interface consists only of a **read** statement that solicits the number of intervals.

```
      program piserial
      double precision h,sum,x,pi,f,a
      integer n

c Define the function
      f(a) = 4.0d0/(1.d0 + a*a)

c Input the number of intervals.
1     print *,' Enter number of intervals:'
      read(5,*,end=100) n

c Calculate the scaling factor
      h = 1.d0/n

c Integrate.  The value of x used to calculate the slice is
c the value at the midpoint of the integration slice.
      sum = 0.d0
      do 10 i = 1,n
         x = h * (dble(i) - 0.5d0)
         sum = sum + f(x)
10    continue
      pi = h * sum

c Output the answer
      print *,' The value of pi for',n,' intervals is',pi
      goto 1
c
c Terminate
100   stop
      end
```

In the parallel version of this program, each node performs a portion of the integration. The decomposition is a domain decomposition that "deals out" the work, as illustrated in Figure 7-2. For example, if you choose 16 nodes and 512 points, each node gets 32 points. The first point goes to node 0, the second point goes to node 1, and so on through the 16th point, which goes to node 15. The 17th point goes to node 0, the 18th point goes to node 1, and so on until all the points have been dealt out. (It is not strictly necessary to deal out the work in this way, because the integration work is evenly balanced. However, since the data is calculated by each node, it is just as easy to deal out as not, and this example deals out the data to give you an example of this technique.)

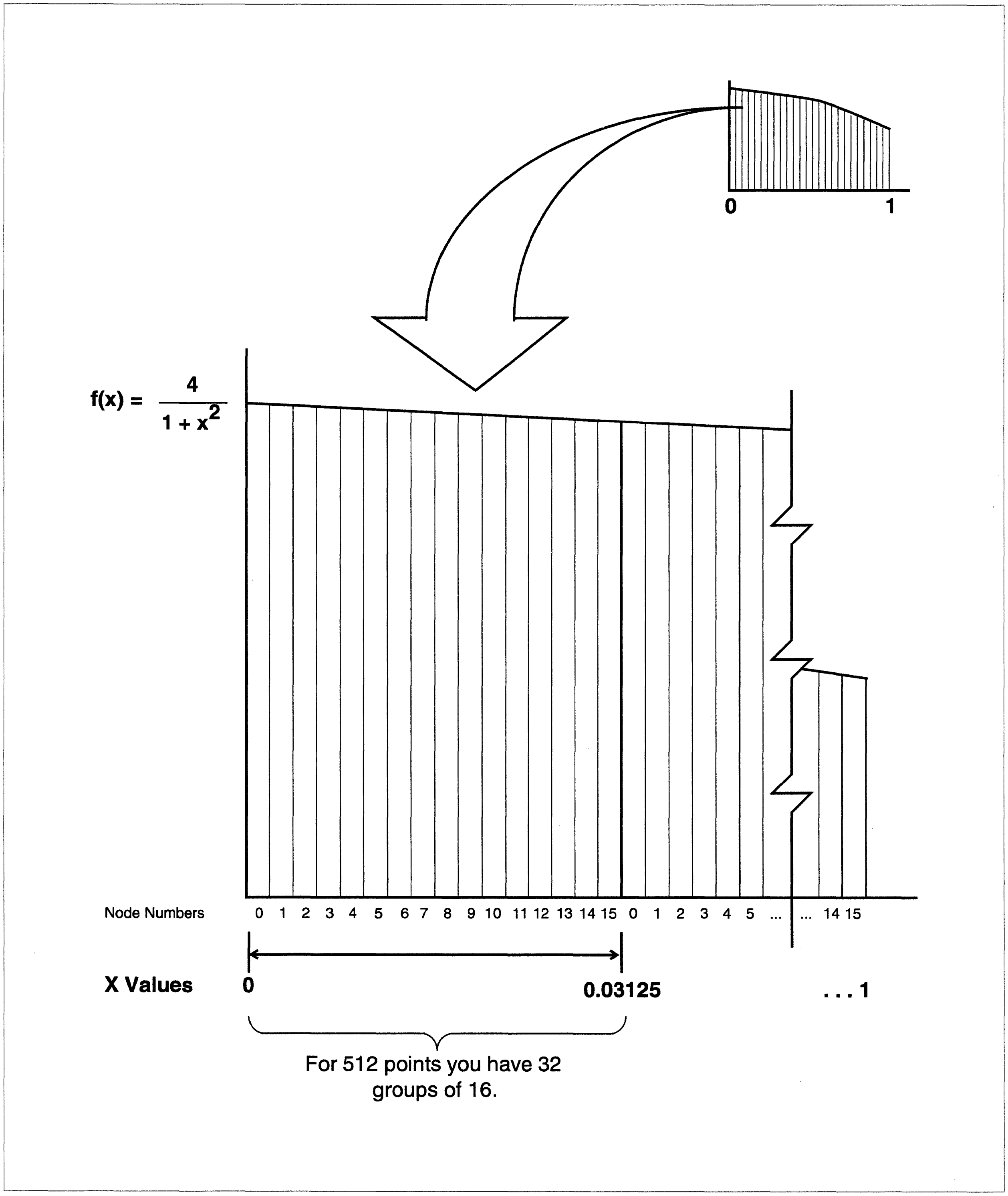


Figure 7-2. The Decomposition Used for the pi Example

Here is the parallel version of the program. The source for this program is available on the Paragon supercomputer in */usr/share/examples/fortran/pi/pinode.f*; differences from the serial version are shown here in **boldface**.

```
      program pinode
      include 'fnx.h'
      double precision h,sum,x,pi,f,a,tmp
      integer n
      integer nodes, iam, intsiz

      data intsiz / 4 /

c Define the function
      f(a) = 4.0d0/(1.d0 + a*a)

c Do some bookkeeping
      iam   = mynode()
      nodes = numnodes()

1     if(iam .eq. 0) then
c        Input the number of intervals.
         print *,' Enter number of intervals:'
         read(5,*,end=100) n
         call csend(300,n,intsiz,-1,0)
      else
         call crecv(300,n,intsiz)
      endif

c Calculate the scaling factor
      h = 1.d0/n

c Integrate.  The value of x used to calculate the slice is
c the value at the midpoint of the integration slice.
      sum = 0.d0
      do 10 i = iam+1,n,nodes
         x = h * (dble(i) - 0.5d0)
         sum = sum + f(x)
10    continue
      pi = h * sum
      call gdsum(pi,1,tmp)

      if(iam .eq. 0 )then
c Output the answer
         print *,' The value of pi for',n,' intervals is',pi
      endif

      goto 1
c
```

```
c Terminate all nodes
100   i = kill(0, 9)
      end
```

Note that the parallel version is not much longer than the sequential version. Note also that the decomposition takes place entirely in the **do** statement. The sequential version is:

```
      do 10 i = 1,n
```

while the parallel version is:

```
      do 10 i = iam+1,n,nodes
```

If you run the application on more nodes, you don't have to change one line of the node program!

In the parallel version, only node 0 interacts with the user. The other nodes do only calculation. If the **print** and **read** statements were not surrounded with **if(iam .eq. 0)then** ... **endif** statements, then when you ran the program on 100 nodes you would have to input the number of intervals 100 times and see the answer 100 times!

Example Application: Matrix*Vector Multiplication

The following example computes the matrix-vector product $y = Ax$, where A is an n x n matrix and x and y are vectors with n components. In addition to the numerical technique, this example illustrates the use of the parallel file I/O calls.

The matrix A is assumed to be too large to fit in the node's memory, requiring an "out-of-core" multiplication. For simplicity, n, the number of rows in the matrix, is assumed to be divisible by p, the number of nodes in the application. The number of rows per node, n/p, is referred to as m.

The problem decomposition is again a domain decomposition. Each node collects all of x, but then takes only a portion of A (specifically m rows) to form its portion of the product vector. There is no attempt to "deal out" the rows of A.

The vector x is initially divided among the nodes. (This example assumes that each node has obtained its portion of x before this routine is called.) Each node contains m components of x. Node 0 has components 1 through m; node 1 has components m + 1 through 2*m, etc. (In general, node Z has components (Z-1)*m through Z*m.) The answer, the vector y, will be stored in the same way.

The matrix A, which is too large to fit in a single node's memory, is also divided among the nodes. It is initially stored in a file called *matrix*. The elements of the matrix are stored in the file by rows, as follows:

A(1,1), A(1,2), ... A(1,n), A(2,1), A(2,2), ... A(2,n), ... A(n,1), A(n,2), ... A(n,n)

Each row of the matrix *A* has *n* elements of length *REALSIZE* bytes, and so each row takes up *n**REALSIZE* bytes in the file. Each node is responsible for *m* rows in the matrix; it reads its portion of the matrix from the file by first moving the file pointer to **mynode()***m**n**REALSIZE* bytes from the beginning of the file, then reading *m* rows of *n**REALSIZE* bytes each beginning at that point.

Here is the code that collects *x*, reads the node's portion of *A*, and performs the multiplication:

```
      subroutine matvmul(m, n, x, y, xtotal, arow)
      integer REALSIZE
      parameter(REALSIZE = 4)
      integer ncnt, fileptr, xlens(128)
      integer m, n
      real x(m), y(m), xtotal(n), arow(n)
c
c  m is n/p where n is the dimension of A
c  and p is numnodes()
c
c  Collect all of x on each node.
      do 3 i = 1, numnodes()
         xlens(i) = m*REALSIZE
3     continue
      call gcolx(x, xlens, xtotal)
c
c  Open the file and seek to the appropriate location

      open(unit=10, file = 'matrix',
     +     form = 'unformatted')
      fileptr = lseek(10, mynode()*m*n*REALSIZE, 0)
c
c  Read the rows and use the BLAS call sdot() to do
c  the multiplication.
      do 10 i = 1, m
         call cread(10, arow, n*REALSIZE)
         y(i) = sdot(n, arow, 1, xtotal, 1)
10    continue
    •
    •
    •
```

This subroutine takes the following parameters:

m The size of each node's portion of the matrix *A* and the vector *x* (*n*/*p*).

n The number of rows and columns in the entire matrix *A* and the number of elements in the entire vector *x*.

x This node's portion of the vector *x* (*m* elements).

y This node's portion of the result vector *y* (*m* elements).

xtotal A temporary array used to hold the entire vector *x* (*n* elements).

arow A temporary array used to hold one row of the matrix *A* (*n* elements).

The subroutine first calls **gcolx()** to collect the nodes' portions of *x* together into the array *xtotal.* It then opens the file containing *A*, moves the file pointer to the beginning of the section of the file that belongs to this node, and then reads *m* rows from the file. After reading each row, it uses the BLAS (Basic Linear Algebra Subroutines) routine **sdot()** to perform the dot product between the current row and the vector *x*, storing the result (a scalar) into the appropriate element of the vector *y*.

NOTE

You must use the **-lkmath** switch on the **if77** command line to link in the library that contains **sdot()**.

See the *Paragon™ System Fortran Calls Reference Manual* for more information on **gcolx()**; see Chapter 5 for information about parallel file I/O; see the *CLASSPACK Basic Math Library User's Guide* or *CLASSPACK Basic Math Library/C User's Guide* for more information on **sdot()**.

Example Application: The N-Queens Problem

This application collects all the board configurations that solve the N-Queens problem. This problem is: "Given an N x N chess board, where can you place N queens so that no queen can capture any other?" In chess, queens attack in straight lines along the X, Y, and diagonal directions.

The N-Queens problem is typical of problems for which there is no analytical solution. Instead, there exists a large set of candidate solutions. You test each solution and accept those that pass.

The difficulty lies in the enormous size of the candidate set. For example, an 8 x 8 chess board has 64 squares. The total number of possible positions for 8 queens can be represented as the *combination* of *n*=64 things taken *m*=8 at a time. The formula for the number of combinations is:

```
n! / ( m!* (n-m)! )
```

which evaluates to 2^{32} possibilities. Even on a state-of-the-art sequential computer, it would take several hours to check every one of those combinations.

Even before you begin thinking about an algorithm, however, you can eliminate a large number of possibilities. For example, any solution that has more than one queen in the same column is invalid. This reduces the number of possibilities to 8^8 or 2^{24}.

This section shows how to use a Paragon supercomputer to evaluate those 2^{24} possibilities. You can arrange the possibilities into a tree. The technique involves following a tree down until it either reaches a dead end (an invalid state) or until it reaches a leaf (a valid solution). Figure 7-3 illustrates such a tree. To make the figure simpler, the chess board is shown as 4 x 4. Instead of 2^{24} possibilities, you have 2^8.

The root of the tree (the zero level) is the null board — no queens present. The next level (the first level) consists of states where a queen is in each of the positions that make up the first column. In Figure 7-3, there are four of those. In an 8 x 8 board, there would be eight.

The next level (the second level) consists of states with two queens on the board, one in the first column and one in the second. In Figure 7-3, there are four of those under each second level state. Notice, however, that some states are already invalid. There is no need to follow the tree any further down this branch. In Figure 7-3, the two leftmost states in the second level are invalid. The second state in the first level has three dead ends in its second level.

You can see how the algorithm is going. Some paths are going to finish early because they reach dead ends. Others are going to take longer and reach the solutions at the leaves. This is a problem for control decomposition.

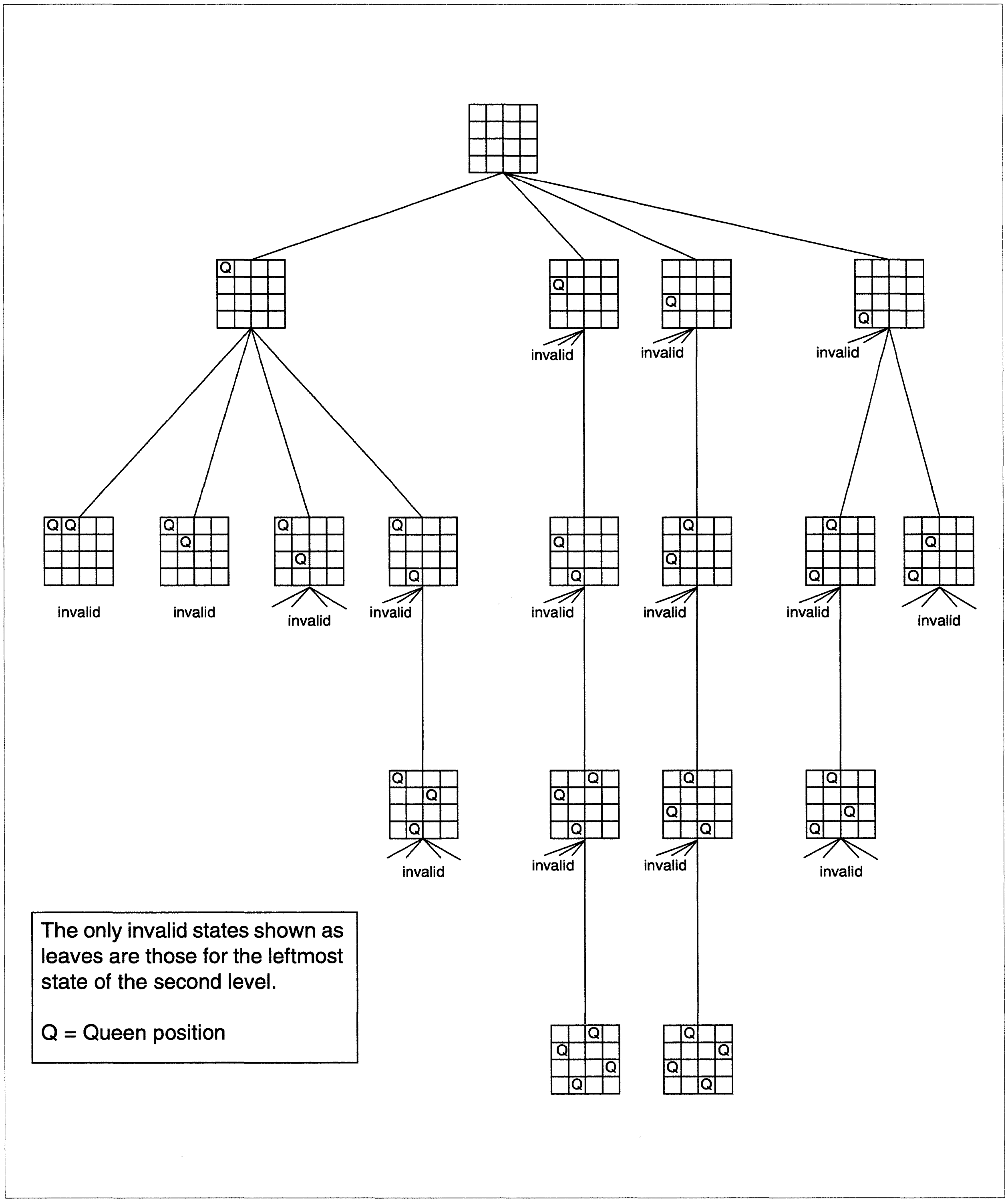
Manager/worker decomposition (a type of control decomposition) is a useful way of achieving balanced computational loads when the application consists of a large number of tasks that are of varying length. Because there is no way of determining up front what the length of the task is, the method consists of dividing the application into a large number of tasks (more than the number of nodes) and then assigning tasks to individual nodes as the node becomes available.

One way of generating the task is for the manager node to follow the tree down until the number of states is larger than the number of available nodes. As a further enhancement, the manager node may even enlist the aid of the other nodes when doing this initial processing.

Then, the manager node assigns a state to a node. The node follows that state down the tree and collects all the possible solutions. When the node finishes, it reports its solutions, if any, and requests more work. In the case of a 4 x 4 board, the tree is shallow and there are only two solutions. An 8 x 8 board results in 92 solutions.

The directory */usr/share/examples/c/nqueens* contains a C version of the 8 x 8 8-Queens problem. The example is written in C because the N-Queens algorithm makes use of recursion.

In this example, a task is represented as a partially-filled board (only the first few columns contain queens) given to one of the nodes. The example as described here runs on four nodes. Node 0 is the manager, and nodes 1 through 3 are the workers. The manager is assigned a certain number of columns (in this example, two) and creates partial boards by placing queens on the board, one for each column it is assigned. When the manager controls two columns of an 8 x 8 board, it creates 64 partial boards.



The only invalid states shown as leaves are those for the leftmost state of the second level.

Q = Queen position

Figure 7-3. The N-Queens Solution Tree for a 4 x 4 Board

Also, in this example, the manager does not create the boards intelligently. For example, the manager will create a board with two queens in the same row. If a worker gets a partial board that contains invalid queens (such as two queens in the same row), the worker immediately throws the board away and requests another.

The manager creates boards by counting in a radix equal to the number of rows in the board. Each digit in the resulting number represents a column with the least significant digit being column 0. The value of the digit is the row position of the queen. Hence, 00 represents two queens in row 0, and 01 represents one queen in row 0 of column 0 and another queen in row 1 of column 1.

The workers signal their availability by sending a "ready" message to the manager. This is a zero length message of type **READY**. When the manager receives a **READY** message, it determines who sent it, then sends a partial board to that node as a message of type **TASK**. The manager keeps doing this until it has no more partial tasks to assign. Finally, the manager waits until all workers are idle (that is, it receives a **READY** message from every worker) and then sends a final message with the special value **FINISHED** to all workers.

Here are the key lines that implement the manager control.

```
/* This is the manager part */
    if (!iam) {      /* If I am node 0 */
      printf("\n\n\n");
      printf("\nSTARTING ...  \n");

/* Manager keeps a count of how many workers are available
and sends out boards to a worker when the worker identifies
itself as READY. The manager uses the routine get_board() to get
a new board. There are no more new boards when this routine
returns DONE.*/

     while ( get_board(board) != DONE ) {
        crecv(READY,NULL,0);
        nodenbr = infonode();
        msgcount++; /* Count how many nodes are ready */
        csend(TASK,board,sizeof(twoD),nodenbr,0);
        msgcount--; /* When a node gets a task, it is no longer
                       ready for another.  Hence, decrease
                       msgcount */
     }

/* Wait for all workers to be free (the msgcount must be equal
   to the number of worker nodes) */

     while(msgcount != nodes-1) {
        crecv(READY,NULL,0);
        msgcount++;
     }
/* Send the FINISHED message to all nodes and then say goodbye */

        board[0][0] = FINISHED;
        csend(TASK,board,sizeof(twoD),-1,0);
        goodbye();
    }
```

The manager does not know if a worker has found a solution or not, and the workers do not know how many initial boards there are. When a worker receives a partial board, it first checks for the special value **FINISHED**, and calls **goodbye()** if it finds this value. (The **goodbye()** routine prints a summary message in the output file, closes the file, and exits.) Next, the worker checks that the queens already on the board are valid. If they are, the worker finds all the solutions that exist with that partial board by recursively calling **move_to_right()**. When the worker finds a solution, it writes the solution to a file called *queens.out*. This file was opened by all nodes in mode **M_LOG** (1), which is the mode in which all nodes have a common file pointer and access the file on a first-come first-served basis.

Here are the key lines that implement the worker control.

```
else {
/* This is the worker part. */

/* Each node enters an infinite loop where it receives a partial
board and checks whether that partial board contains valid
queens.  If the board contains a FINISHED message, the node
cleans up and exits by calling goodbye().  If the board contains
invalid queens, the node considers itself done with the task.
Otherwise, it tries to place a queen in the next column by calling
move_to_right().  This routine will find all possible solutions
given the initial board. */

        for(;;) {
            csend(READY,0,0,0,0);
            crecv(TASK,board,sizeof(board));
            if(board[0][0] == FINISHED) {
                goodbye();
            }
            if ( chk_board(board) ) {
                move_to_right(board,0, MCOLS);
            }
        }
    } /* end of else */
```

There are many opportunities for optimizing this algorithm. For example, you could write the manager in such a way that it only gave workers boards that had the potential of containing one or more solutions. In addition, the manager could mark positions on the board that are invalid due to the presence of the initial queens, and the worker would not have to check those.

The file *queens.out* contains copies of all the 92 solutions for the 8-Queens problem. Each board is preceded by a header that identifies the node that found the solution and the number of solutions found so far by the node. Finally, the total number of solutions is printed. The tail of the file looks as follows:

```
               •
               •
               •
Node 1 found solution 30

  0 1 2 3 4 5 6 7
0 - - - Q - - - -
1 - - - - - Q - -
2 - - - - - - - Q
3 - - Q - - - - -
4 Q - - - - - - -
5 - - - - - - Q -
6 - - - - Q - - -
7 - Q - - - - - -

Node 2 found solution 31

  0 1 2 3 4 5 6 7
0 - - - - Q - - -
1 - - - - - - Q -
2 - - - Q - - - -
3 Q - - - - - - -
4 - - Q - - - - -
5 - - - - - - - Q
6 - - - - - Q - -
7 - Q - - - - - -

Node 3 found solution 31

  0 1 2 3 4 5 6 7
0 - - - - Q - - -
1 - - Q - - - - -
2 - - - - - - - Q
3 - - - Q - - - -
4 - - - - - - Q -
5 Q - - - - - - -
6 - - - - - Q - -
7 - Q - - - - - -

Total solutions = 92
```

If you want to investigate another manager/worker application, look at the *triangle* program in */usr/share/examples/c/triangle*. Its operation is described in a *README* file.

Improving Performance 8

Introduction

This chapter presents some techniques you can use to improve the performance of your parallel applications. It includes the following sections:

- Single Node Performance.
- Multi-Node Performance.
- I/O Performance.

In general, however, the best thing you can do to improve performance is to choose an efficient numerical method and algorithm for solving your problem. A good numerical method and an efficient algorithm will always give better performance than a poor method and algorithm. This is true even if the good method is implemented in a high-level language and the poor method is implemented in hand-coded assembly language.

Another general performance technique is to use the Paragon system's profiling and performance analysis tools to help pinpoint the parts of your application that could benefit the most from optimization. See the *Paragon*™ *System Application Tools User's Guide* for information on the available tools.

Single Node Performance

This section discusses things you can do to increase the speed of calculation (MFLOPS or GFLOPS) on each node. Many of these are general performance-improvement techniques that you can use on any computer; some of them are specific to the i860® microprocessor. Techniques discussed in this section include:

- Use profiling tools.
- Avoid repeated use of system calls.
- Avoid virtual memory paging.
- Use compiler optimizations.
- Increase problem size.
- Access contiguous memory locations.
- Use caching wisely.
- Use optimized libraries.
- Use assembly language subroutines.
- Avoid error checking (C language only).

Use Profiling Tools

The Paragon system comes with the **prof** and **gprof** profilers, and their graphical versions **xprof** and **xgprof**. You can use these tools to help track down the parts of your application that are consuming the most time and concentrate your optimization efforts there. See the *Paragon™ System Application Tools User's Guide* for more information on these tools.

Avoid Repeated Use of System Calls

Don't make a system call twice if once will do. This is an obvious performance improvement technique, but unfortunately it is missing from many applications. For example, a process may need its node number and process type to do message passing. Avoid using **mynode()** and **myptype()** each time you need those numbers. Instead, invoke each once and store their values in variables.

Avoid Virtual Memory Paging

The operating system provides *virtual memory*, which lets you use more memory than is physically available on the node. When a program tries to allocate a memory space that is larger than the node's available free memory, one or more 8K-byte *virtual memory pages* that haven't been referenced recently are *paged out*. This means that their contents are written to disk and replaced with the new data. Later, when the program references data in the paged-out memory section, a different section is paged out and the old data is *paged in* (read back from disk) in its place.

Although virtual memory makes it possible for the system to support multiple users and very large programs, you should try to avoid it when you can. Accessing pages of virtual memory that are not currently paged in is much slower than physical memory and generates a lot of disk activity. Try to reduce the memory used by your application until it fits in physical memory, including dynamically-allocated buffers and system message buffers (see "Understand Message-Passing Flow Control" on page 8-13 for information on the sizes of system message buffers).

You can use the **vm_stat** command to get information about your application's memory usage. See the *Paragon™ System Commands Reference Manual* for information on this command.

Once you have reduced your application so that it fits in physical memory, you may be able to use the **-plk** switch to lock parts of your application into physical memory. This reduces paging and improves message-passing latency, but has certain consequences; see "Process Locking" on page 8-15 for more information.

Use Compiler Optimizations

When you compile a program, you can use *compiler optimization switches* to tell the compiler what techniques to use to optimize your code. Optimization can produce a compiled program that does the same work in less time by making better use of the processor's special features. However, optimization can sometimes produce a program that runs more slowly or produces wrong answers, so it must be used carefully.

The compiler optimization switches you can use and compiler-specific code changes you can make depend on which language you program in and the revision level of the compiler. See the *Paragon™ System Fortran Compiler User's Guide* or *Paragon™ System C Compiler User's Guide* for complete information on compiler optimizations for your specific compiler. However, here are a few general hints:

- Experiment with the **-O** switch, which controls the level of compiler optimization:

  - Level 0 performs no optimization.

  - Levels 1 and 2 perform straightforward optimizations that should always result in improvement.

  - Levels 3 and 4 attempt to make use of the i860® microprocessor's *pipelining* and *dual-instruction* modes to improve performance; whether or not they improve your program's performance, and by how much, depends on the characteristics of your program.

  In some cases, different parts of the program should be compiled with different optimization levels.

- Try the **-Mvect** switch to invoke the *vectorizer*. The vectorizer attempts to rearrange your code to allow more efficient use of pipelining. You can get better results out of the vectorizer if your innermost loops have the following characteristics:

  - The loop index increments the first dimension in Fortran, or the last dimension in C.

  - Arrays are accessed with unit stride.

  - The number of iterations within the loop is not too small.

  Also, **if** tests and subroutine calls should be avoided within the loop. See the *Paragon™ System Fortran Compiler User's Guide* or *Paragon™ System C Compiler User's Guide* for specific examples of code changes you can make.

- If your application does not depend on strict IEEE semantics for mathematical operations, try the **-Knoieee** switch. This switch provides much faster mathematical operations than those provided by the default IEEE math library, but may result in slightly decreased accuracy and different behavior in exceptional circumstances (operations on 0 or infinity and NaNs).

- Use the **-Mnostride0** switch, unless your program accesses arrays with zero stride (that is, incrementing the array pointer by 0 in each loop iteration). There are some important compiler optimizations that are only possible if the compiler knows the code does not do this.

Increase Problem Size

Once you have optimized a program's single-node performance, you may find that running the program on a larger number of nodes with the same data set gives a lower per-node performance. This can occur because the per-node vector size has gone down, reducing the efficiency of the i860 microprocessor's pipelines. To avoid this problem, you can increase the problem size as you increase the number of nodes, or you could even write two different inner loops—one optimized for short vectors, the other optimized for longer vectors.

Access Contiguous Memory Locations

Whenever you access memory, try to access contiguous memory locations. In particular, whenever your program reads or changes the value of an array element in memory, try to be sure that the next array element it reads or changes is adjacent to the previous one in memory. This is important because the i860 microprocessor accesses memory in 4K-byte *physical memory pages*. Once you have read from a physical page, another read from the same page takes only one more cycle, but a read from a different page takes 10 to 14 more cycles. Every cycle spent switching from one page to another is a cycle that can't be used for calculation.

To keep your memory accesses within a physical page, you can use some of the following techniques:

- Group a series of memory reads out of the same array.

- Do consecutive references across the rows (C) or down the columns (Fortran) of matrices. (In C the rightmost index of an array varies the fastest, while in Fortran the leftmost index varies fastest. This means that, for example, if you distribute the elements of a two-dimensional array among the nodes, you should give out rows in C and columns in Fortran.)

- "Strip-mine" loops, so that you do several accesses to the same array at a time. For example, you should read several elements from vector *A*, then several from *B*, instead of reading *A[1]*, *B[1]*, *A[2]*, *B[2]*, and so on.

You should also try to group reads and writes. Once you have read from a page, a following write takes about 6 more cycles than a following read. Switching from write to read also takes about 6 cycles.

Use Caching Wisely

The i860 microprocessor has a 16K-byte *data cache* for recently-accessed memory locations. Whenever you read or change a bit in memory, a 32-byte area of memory containing that bit is copied into the cache. (This 32-byte area is called a *cache line* and always begins on a 32-byte boundary.) When you access memory that is already in the cache, the access is very fast. However, whenever a new cache line is copied into the cache, cache lines that have not been accessed recently are written back to memory (if necessary) and removed from the cache to make room. Try to arrange your code so that all operands for a loop can be accommodated in the cache at the same time.

The i860 microprocessor also has an *instruction cache* (4K bytes on the i860 XR, 16K bytes on the i860 XP) which is used to hold program instructions once they have been fetched and decoded from memory. You can use this cache in two ways:

- Try to keep your loops small. If the code for an entire loop fits in the instruction cache, the loop can execute very quickly.

- In an **if/else** block, try to put the code that is used more often in the **if** part and the code that is used less often in the **else** part. The instruction cache works in a "lookahead mode," and when pre-fetching instructions will fetch the code immediately following the **if**. If the **else** branch is executed instead, the **if** branch code must be discarded from the cache.

Both these techniques can be used in high-level languages as well as assembly code.

Note that the data cache, the instruction cache, the physical memory page, and the virtual memory page are separate functions that have different sizes and different effects. In general, cache management is handled by the compiler, but you should try to arrange your code to make the compiler's job easier.

Use Optimized Libraries

Several optimized libraries of math and utility functions are available with the operating system. These libraries have been carefully hand-tuned to give the best possible performance; you can save time and increase efficiency by using routines from these libraries rather than writing the equivalent code yourself. The available libraries include:

- The Basic Math Library (*libkmath.a*). This library is a standard part of the operating system; it includes optimized BLAS (Basic Linear Algebra Subroutines) and FFT (Fast Fourier Transform) routines. See the *CLASSPACK Basic Math Library User's Guide* or *CLASSPACK Basic Math Library/C User's Guide* for more information.

- The Signal Processing Library (*libsignal.a*). This library is an optional product; it includes optimized vector and signal-processing routines. See the *CLASSPACK Signal Processing Library User's Guide* or *CLASSPACK Signal Processing Library/C User's Guide* for more information.

Note that these are *single-node libraries*; they improve the numeric performance of each node of your program, but do not affect its multi-node performance.

Use Assembly Language Subroutines

Re-writing key routines in the i860 microprocessor's assembly language can sometimes bring significant performance benefits. See the *Paragon™ XP/S System i860™ Microprocessor Assembler Reference Manual* for information on the assembler.

Avoid Error Checking (C Language Only)

In C, there are two versions of most calls in *libnx.a*: the standard version and the *underscore version* (for example, the underscore version of **crecv()** is **_crecv()**). When you call the standard version, the call checks for certain error conditions before it returns; if an error is detected, the call terminates your program with an error message. The underscore version works the same as the non-underscore version, but if an error occurs, the call simply returns the value -1 and sets the external variable *errno* to a value that describes the error. This is useful if you want to handle an error yourself and not let the system do it. But if you are confident that your program is working, you may choose to use the underscore version and *not* check the return value, thereby improving performance. (If an error *does* occur, unexpected and difficult-to-debug behavior will result, so use this technique with caution.)

Multi-Node Performance

This section discusses things you can do to increase the efficiency of applications running on multiple nodes, including:

- Use dynamic memory allocation for large arrays.
- Avoid serializing calls.
- Use ParaGraph.
- Maintain data locality.
- Overlap computation and communication.
- Avoid message buffering.
- Align application buffers.
- Understand message-passing flow control.

Use Dynamic Memory Allocation for Large Arrays

You should always use *dynamic memory allocation* for large arrays. Dynamic memory allocation means allocating the memory for the array at run time, using the **ALLOCATE** statement in Fortran or the **malloc**() call in C. The alternative, *static* memory allocation, means declaring the array in the program source.

Dynamic allocation is important even on one node, but becomes more and more important as the number of nodes increases. The larger the array and the more nodes, the more performance can be improved by using dynamic memory allocation. If the array or number of nodes is large enough, the application may not run at all unless you use dynamic memory allocation.

For example, the following program fragment uses static memory allocation. It simply creates two 4M-byte arrays of **real*4** (one in a common block, the other not) and initializes each element of each array to the element number.

```
      parameter huge_size = 1024*1024
      real*4 huge(huge_size), huge_common(huge_size)
      common /giant/ huge_common
      integer i

      do 10 i=1,huge_size
          huge(i) = i
          huge_common(i) = i
10    continue
```

The equivalent code with dynamic memory allocation is as follows (changes are shown in boldface). With 16M bytes of memory on each node, this version runs as much as *ten times as fast* as the previous version on one node; it runs as much as *fifteen times as fast* on eight nodes. The more nodes, the greater the speedup.

```
      parameter huge_size = 1024*1024
      real*4 huge(huge_size), huge_common(huge_size)
      pointer(p, huge)
      common, allocatable /giant/ huge_common
      integer i

      allocate(huge, /giant/)

      do 10 i=1,huge_size
          huge(i) = i
          huge_common(i) = i
10    continue

      deallocate(huge, /giant/)
```

Note that a common block must be declared **ALLOCATABLE** before it is allocated with an **ALLOCATE** statement. A variable or array that is not part of a common block must be declared as a pointer-based variable with a **POINTER** statement before it is allocated with **ALLOCATE**; the corresponding pointer variable, in this case **p**, does not have to be used. See the *Paragon™ System Fortran Language Reference Manual* for more information on these statements.

The reason statically allocated arrays cause your program to run slowly is that, since they are compiled into the program, the initial contents of the array must be obtained from the executable program on disk. Whenever a process on a compute node reads or changes a byte in a memory page of statically allocated data that it hasn't touched before, the data for that page may have to be *paged in.* (See "Avoid Virtual Memory Paging" on page 8-3 for an introduction to virtual memory paging.) A message requesting the initial contents of that page is sent to the node in the service partition where the compute process's parent process is running. The entire page—8K bytes—is then sent back to the compute process. This occurs even if the statically allocated data is uninitialized (all zeroes).

Sending these pages across the mesh takes time. Even worse, if many compute processes all want pages at the same time, the parent process's node can become overwhelmed, slowing the application drastically. The effect is magnified if the statically-allocated array is so large that parts of the operating system have to be paged out to make room for it; in this case, pages of the operating system have to go out at the same time pages of static data are coming in.

When you use **ALLOCATE** or **malloc()** to dynamically allocate an array, the memory is not associated with the program on disk. Instead, each node has its own copy of the array, and it doesn't have to be paged in. When a compute process touches a page of dynamically-allocated data it hasn't touched before, the page is simply allocated from the available node memory—no messages are sent. This greatly reduces traffic on the mesh and increases the performance of your application.

Avoid Serializing Calls

Avoid using serializing calls repeatedly or on many nodes. A *serializing* call is one that relies on a single resource which can only service one request at a time (typically a daemon or server on the boot node). Using a serializing call once takes little time, but if many nodes in a large application call it at the same time the boot node can only service these requests one at a time. Each node must wait until the boot node services its request, which can cause the entire application to run slowly.

Many calls that perform I/O or make use of the file system, such as **stat()**, **chdir()**, and **chmod()**, are serializing calls, because they must communicate with the root file system server on the boot node. **getrusage()** is also a serializing call, because it sends a message to all the I/O nodes to get information on the caller's I/O activity. You can detect the presence of serializing calls in your program by profiling it. If common operations, especially I/O operations, are taking much more time than expected, they may be serializing calls.

Whenever possible, avoid overuse of serializing calls by having only one node make the call. For example, instead of having every node process call **stat()**, have one node call **stat()** and then use **gisum()** to distribute the information to the other nodes. Also, instead of having every node process call **chdir()** when it starts up, have the controlling process call **chdir()** before creating the node processes. You can avoid serialization in I/O by using the I/O mode **M_RECORD** or **M_GLOBAL**, as described under "Use the Appropriate I/O Mode" on page 8-24.

Use ParaGraph

The Paragon system comes with the ParaGraph performance visualization tool. You can use this graphical tool to help analyze your application's message passing behavior and determine where to concentrate your optimization efforts. See the *Paragon™ System Application Tools User's Guide* for more information on ParaGraph.

Maintain Data Locality

Wherever possible, try to distribute the data to the nodes so that each node has all the data it needs, and does not have to get any data from other nodes. Where it is not possible to keep all related data on one node, try to keep the data as close as possible to the nodes that need it.

For example, suppose you are writing a simulation where the value of each data point in a plane is computed from the values of nearby data points. To parallelize this simulation, you would divide up the plane into segments and assign each segment to a node. However, each node must communicate with other nodes to get data for points that are just past the edge of its data segment. Since the Paragon system has a mesh architecture, you would typically divide up the plane in a two-dimensional decomposition (rectangles). You would then assign the rectangles to nodes in such a way that neighboring segments are the responsibility of neighboring nodes. (Use the **nx_app_rect()** call to determine the "shape" of your application.) This will reduce message traffic and ensure that each message reaches its destination as quickly as possible.

Overlap Computation and Communication

The message-passing system calls are available in both *synchronous* versions (call names beginning with **c**) and *asynchronous* versions (call names beginning with **i**). Synchronous calls do not return until the message-passing operation is complete; asynchronous calls return immediately, giving you a message ID that you can use to check when the operation is complete.

Although the synchronous calls are easier to use and have slightly lower overhead, you should use the asynchronous calls whenever the results of the call are not needed immediately. Using asynchronous calls can let your application do useful computation in the time when it would otherwise just be waiting for a message to arrive. During this time, the node's message coprocessor can process the communication without interrupting the main processor.

Avoid Message Buffering

Try to avoid message buffering whenever possible. For example, assume that you have the same process running on two nodes and that these processes must exchange information. Each process must issue a receive and a send. If a **csend()** call is executed before its corresponding **crecv()**, the message is sent and buffered in a system buffer (if there is not enough space in system buffers, the sender blocks). When the **crecv()** is executed, the message is copied from the system buffer into the application buffer. (For detailed information on message buffering, see "Understand Message-Passing Flow Control" on page 8-13.)

Your code runs more efficiently if you can avoid the system buffer and copy the message directly into the application buffer. You can do this by using an **irecv()** (the asynchronous receive) and posting the receive before the corresponding **csend()**. Remember, however, that because the nodes do not run in lock step, coding the **irecv()** before its corresponding **csend()** does not guarantee that the **irecv()** is executed before its **csend()** (even if the same program runs on every node). You can make sure that the **irecv()** is executed before the **csend()** by using zero-length messages to synchronize the nodes, as shown in the following example.

For example, consider the following C routine, *shadow*. This routine might appear in an application that needs to have the nodes exchange the rows of a matrix (a Fortran version would probably exchange columns instead).

A typical application for *shadow* might be a Gauss-Seidel iteration or any technique based on nearest neighbor interactions. The application processes a two dimensional array called *s*[ ][ ] and exchanges rows between nodes. The first index in the array represents a row, and it is passed as a pointer. Each node contains a horizontal partition of the array with *range* rows. It has a top buffer (*s*[0]) and a bottom buffer (*s*[*range*+1]) containing the boundary rows from other nodes.

```
void shadow(
   long topnode;
   long botnode;
   int (*s)[MAX_LATTICE]
   int range)
{
   long topid, botid, syncbotid, synctopid;

   /* Node sends upper boundary row s[1] to the bottom buffer
   s[range+1] of the node controlling the upper partition,
   (topnode).

   Node sends lower boundary row s[range] to the top buffer s[0]
   of the node controlling the lower partition, botnode */

   topid = irecv(TOP,  s[0],       sizeof(s[0]));
   botid = irecv(BOT,  s[range+1], sizeof(s[range+1]));
```

```
    /* The following code ensures that the csend()s corresponding
    to the above irecv()s are not executed until all the irecv()s
    have been posted. */

    syncidtop = irecv(SYNCTOP, 0, 0);
    syncidbot = irecv(SYNCBOT, 0, 0);
    csend(SYNCTOP,0,0,topnode,0);
    csend(SYNCBOT,0,0,botnode,0);
    msgwait(syncidtop);
    msgwait(syncidbot);

    /* End of synchronization code. */

    csend(BOT, s[1],     sizeof(s[1]),     topnode, 0);
    csend(TOP, s[range], sizeof(s[range]), botnode, 0);

    msgwait(topid);
    msgwait(botid);
}
```

Note that when the data sends are performed the asynchronous receives have already been executed. This is ensured by the zero-length synchronization messages. The data then goes directly into the application buffer (unless it is paged out, as discussed later in this section).

Another way of achieving synchronization is to issue a **gsync()** after the **irecv()**'s. However, **gsync()** can be expensive—it synchronizes *all* the nodes, when all that's really necessary is to synchronize senders and receivers. A good rule of thumb is to synchronize only what is really necessary.

Align Application Buffers

Try to ensure that send and receive buffers are properly aligned and sized whenever possible. Although the message-passing system calls will work with any size or alignment of buffers, the hardware works best with well-aligned buffers. The software may have to copy messages that are in misaligned buffers to new, aligned buffers, which decreases performance. There are several degrees of alignment. All other things being equal:

1. The best performance can be achieved by aligning the send or receive buffer on a 4K-byte boundary (this means that the buffer's address is an even multiple of 4K). This corresponds with the i860 microprocessor's 4K-byte physical memory page.

2. Good performance (only slightly worse) occurs if the buffer is aligned on a 32-byte boundary, which corresponds with the microprocessor's cache line, but crosses a 4K-byte memory page boundary.

3. The next-best performance (not nearly as good) occurs if the buffer is aligned on an 8-byte boundary, which corresponds with the microprocessor's FIFO size.

4. Some performance improvement can be seen if the buffer is aligned on a 4-byte boundary.

5. The worst performance comes when the buffer is not even aligned on a 4-byte boundary (that is, its address is not a multiple of 4).

To be sure the buffer is well-aligned, you should use **malloc()** to allocate it rather than allocating it statically. (This call is available in both Fortran and C.) Buffers allocated with the standard **malloc()** or its derivatives will always be aligned on a 32-byte boundary.

You can arrange for the pointer to a message buffer to be on a 4K-byte boundary by using pointer arithmetic. This technique can be used even if the buffer is statically allocated. You do this by declaring your buffer to be 4095 (4K–1) bytes longer than needed, adding 4095 to the buffer pointer, and then ANDing the pointer with the NOT of 4095. For example, assume that you need a 50K-byte buffer called *buf*. You can cause the buffer pointer *bufp* to be on a 4K-byte boundary by declaring *buf* to be 54K–1 bytes and doing the following pointer arithmetic:

```
char *bufp, buf[55295];
       •
       •
       •
/* bufp points to the nearest 4K-byte boundary in the buffer */
bufp = (char *) (((int)buf + 4095) & ~4095);
```

Finally, when you set up C structures that you intend to use as messages, try to minimize *padding* (open areas within the structure inserted by the compiler to make the following structure element properly aligned). To do this, you should be aware of the sizes and alignments of the different data types; in general, you can minimize padding by placing the larger data types first. Reducing padding reduces the number of empty bytes sent with the message.

Understand Message-Passing Flow Control

Whenever you send a message, the sending process and the receiving process use *message-passing flow control* to make sure that the message is safely stored in memory when it arrives at the destination node. This flow control guarantees that messages flow from their source to destination without blocking on the mesh. If you understand flow control, you can select the message sizes, message-passing configuration parameters, and message synchronization techniques that give the best possible message-passing performance for your application.

Note, though, that the most important thing you can do to improve message-passing flow control is to avoid message buffering (as discussed under "Avoid Message Buffering" on page 8-11). You should only read this section if you cannot avoid message buffering and need to improve its efficiency, or if you really want to understand all the nitty-gritty details of message-passing flow control. If neither of these applies to you, skip to "Recommendations" on page 8-21.

Overview of Message-Passing Flow Control

Here's an overview of what happens when you send a message. (This is a simplified view; the low-level details of message-passing flow control are proprietary and subject to change.)

1. The sending process checks to see if the memory page containing the message to be sent is currently in physical memory. If not, it is paged in. (See "Avoid Virtual Memory Paging" on page 8-3 for information on paging.)

2. The sender checks to see how much memory it thinks is available in system message buffers on the receiver and sends the appropriate number of bytes (see "System Message Buffers" on page 8-16 for details). If the whole message has been sent at this point, the send is complete; otherwise, it waits for a request from the receiver for the next part of the message. This waiting may or may not block the sending process, depending on whether the sending call was synchronous or asynchronous.

3. When the message (or the first part of the message) arrives at the receiving node, the node's operating system checks to see if there is a receive posted for a message of that type by the specified receiving process. ("Posted" means that the process has an outstanding message-receiving call that has not yet been fulfilled.)

    A. If there is a receive posted and the application buffer (the buffer specified in the receive call) is currently paged in, the message is stored directly into the application buffer.

    B. If there is a receive posted and the specified application buffer is not paged in, the message is stored in a system buffer. Then the application buffer is paged in and the message is copied into it.

    C. If there is no receive posted, the message is stored in a system buffer and the receiver waits until a receive is posted. When the receive is posted, the specified application buffer is paged in (if necessary) and the message is copied into it.

4. If the whole message has been received at this point, the receive is complete; otherwise, it sends a request to the sender for the next part of the message and waits. (This waiting may or may not block the receiving process, depending on whether the receiving call was synchronous or asynchronous.) The request also includes the current free space in system message buffers on the receiver, which is used to calculate how big the next part should be. Go back to step 1 and continue until the message has been completely sent and completely received.

Special case: if the sender thinks the message is too large to send all at once, and there is no receive posted, but there is actually enough space in system buffers on the receiver to accommodate the entire message, the receiver stores the first part of the message in a system buffer and immediately sends a special request to the sender saying "send the whole rest of the message." In this case, the entire message can be sent before the receive is posted. Otherwise, only the first part of the message is sent and the rest of the message waits on the sender until the receive is posted.

Process Locking

The message-passing flow control procedures check to make sure that application buffers are paged into physical memory, and copy information from one buffer to another if they are not. You can avoid these steps by using the **-plk** switch on the application command line. This switch locks parts of each process into physical memory, like the OSF/1 system call **plock()** (see the *OSF/1 Programmer's Reference* for information on **plock()**). This locking is also referred to as *wiring*.

The **-plk** switch locks the following parts of your application into physical memory:

- The entire *data segment* (the part of memory that contains global variables) is locked. This occurs when the program is loaded.

- If you use an application buffer that is located on the *stack* (the part of memory that contains local variables) or on the *heap* (the part of memory that is allocated by **malloc()** or **ALLOCATE**), the area from the beginning of the stack/heap to the end of the buffer is locked. This occurs the first time you use the buffer in a message-sending or -receiving call.

All areas of memory not mentioned in this list, including the code segment (the part of memory that contains executable instructions), are not locked and are still subject to paging. Note that locking is done a page at a time: to lock a single byte, the system must lock the entire 8K-byte virtual memory page containing that byte.

The **-plk** switch greatly reduces the effect of virtual memory on your application and improves message-passing latency. However, it has the following consequences:

- Your application must fit in physical memory. If it does not, any operation that results in the allocation or locking of more memory may fail unexpectedly, possibly terminating the application.

  Ideally, the operating system and the application's code and data should all fit into the node's physical memory at once. However, the code segment is subject to paging even when **-plk** is in effect, so the application may still work if there is enough physical memory left over after subtracting the size of the operating system and the total amount of locked data. The definition of "enough" depends on the application's pattern of access to its code in memory and how much of the code needs to be present in memory at once. (See the *Paragon™ System Software Release Notes for the Paragon™ XP/S System* for information on how much memory is needed by the operating system.)

- The physical memory available for other processes is reduced by the size of your application's locked data for the life of your application.

  When **-plk** is in effect, none of the locked data will be removed from physical memory until the application terminates. Even if your application is not actually executing (for example, because it is "rolled out" by gang scheduling), it still retains control of this memory, and the application that *is* currently executing cannot use the memory that is locked by your application. This can cause the other application to run very slowly or "thrash."

To prevent this "thrashing," your system administrator can configure the system so that **-plk** cannot be used in gang-scheduled or standard-scheduled partitions. If **-plk** is allowed at your site, it should be used with extreme care because of its impact on other users.

-plk also conditions message-passing flow control to run more efficiently by assuming that all message buffers are locked into memory. For example, suppose a message is too large to send all at once. With **-plk**, after the first part of the message arrives, the receiver can request the entire rest of the message—no matter how big it is—as soon as the receive is posted. Since the application buffer cannot be paged out, the receiver can be sure it will be there to receive the rest of the message when it arrives. Without **-plk**, the application buffer could be paged out while the second part is on its way, so the second and subsequent parts of the message must be smaller than the available system message buffer space. This means that many more exchanges might be required before the message is completely received.

Packetization

Messages from one node to another may be broken into smaller messages, called *packets*, before they are placed on the mesh. Using packets results in a slight additional overhead on large messages, but it gives much better overall message bandwidth because it allows several large messages to be interleaved on the same wire at the same time.

The maximum size of each packet is referred to as *packet_size*, and is 8192 bytes for MP systems and 1792 for GP systems, by default (this size does not include the header appended to each packet). If a message is larger than *packet_size*, it is sent in several pieces, each at most *packet_size* bytes long. You can change the packet size with the **-pkt** switch on the application command line.

System Message Buffers

In each node process, an area of memory is set aside for *system message buffers*. These buffers are used to store messages that arrive at the node before the receiving process is ready to receive them. For example, if a sending process calls **csend()** before the receiving process has called the corresponding **crecv()**, the message goes into a system buffer in the receiving process. Then, when the receiving process does call **crecv()**, the message is copied from the system buffer to the buffer specified in the **crecv()** call, which is referred to as the *application message buffer.*

The size and behavior of the system message buffers are controlled by several parameters that you can set on the application command line. The following list describes these parameters and their effects.

- The total amount of memory allocated to system message buffers in each process is referred to as *message_buffer*. The *message_buffer* is always *wired* into physical memory, which means that it can never be paged out (see "Avoid Virtual Memory Paging" on page 8-3 for information on paging). This is necessary to ensure that all messages that arrive at the node can be stored somewhere, even if the rest of the application is paged out. The default value of *message_buffer* is 1152K bytes. You can change this size with the **-mbf** switch on the application command line

- The *message_buffer* is divided into an area for messages from any process, and a series of areas dedicated to messages from particular processes. The number of dedicated areas is referred to as *correspondents*, and the size of each dedicated area is referred to as *memory_each*. When a message is received, it is stored in the open area if there is room; the *memory_each* areas are used only when the open area is full. The default value for *correspondents* is **numnodes()**; you can change it with the **-noc** switch. The default value of *memory_each* is determined by the current values of *correspondents*, *message_buffer*, and *packet_size*; you can change it with the **-mea** switch.

- Each node process maintains a value called the *send_avail* for each other process in the application. The *send_avail* is the maximum amount of memory that the process can depend on to be available in its *memory_each* segment in that other process. (The *send_avail* may be smaller than the actual amount of memory available, but is never larger.) When a process sends a message to another process, it decreases the *send_avail* for that process by the message size; when the message has been "consumed" (completely placed in the application buffer on the receiving process), the receiver tells the sender and the sender increases its *send_avail* for the receiving process accordingly. The initial value of *send_avail* is *memory_each*; the *send_avail* value is maintained dynamically by each process and cannot be set on the command line.

- When a process has a large message to send, it uses its *send_avail* value for the receiving process to determine how much of the message to send at first. Two parameters called *send_threshold* and *send_count* control this behavior:

  1. If the *send_avail* value is equal to or greater than the size of the message, the sender sends the whole message at once.

  2. Otherwise, if the *send_avail* value is equal to or greater than the *send_threshold*, the sender sends the first *send_count* bytes of the message and waits for an acknowledgment from the receiver that they have been consumed before proceeding.

  3. Otherwise, if the *send_avail* value is equal to or greater than the *packet_size*, the sender sends the first packet of the message and waits for an acknowledgment from the receiver that it has been consumed before proceeding.

  4. If the *send_avail* value is less than *packet_size*, the send blocks until the receiver tells it that some messages have been consumed and *send_avail* can be increased. (See the discussion of *give_threshold* later in this section for more information on how this occurs.) This blocking may or may not block the sending process, depending on whether it used a synchronous send (such as **csend()**) or an asynchronous send (such as **isend()**).

     Note that deadlock can occur when *send_avail* is less than *packet_size*. For example, suppose that node A and node B's system message buffers are both full. Under normal circumstances, eventually a receive would be posted and the buffered messages would be consumed. But if the two nodes try to exchange messages with synchronous calls, they deadlock: A blocks waiting for more space to become available on B, and B blocks waiting for more space to become available on A.

The default value for *send_threshold* and *send_count* is half of *memory_each*; you can change these two parameters with the **-sth** and **-sct** switches respectively.

- When a message is consumed, the receiver normally informs the sender that the space occupied by the message is available for new messages by "piggy-backing" information on other messages going to the sender. However, if there are no such messages, the sender can get out of date and stop sending messages because it thinks there is no free memory left for it on the receiver. In this case, a parameter called the *give_threshold* comes into play. If a receiver knows that a sender thinks it has less than *give_threshold* bytes of memory free, but there is really more memory available, it sends a special message to the sender telling it how much memory is really available. The default value for *give_threshold* is *packet_size*; you can change it with the **-gth** switch.

Message-Passing Configuration Switches

The switches that control the message-passing configuration parameters discussed earlier in this section are referred to as the *msg_switches*. Although the default values of these parameters have been chosen to give good results for "typical" applications, you may be able to improve your application's message-passing performance by using different values.

You use the *msg_switches* on the command line of a parallel application. These switches override the default values of the specified parameters for that run of the application; they do not have any effect on other runs or other applications.

If the application was linked with the **-nx** switch, the *msg_switches* are automatically interpreted and removed from the command line before the application starts up. An application linked with **-lnx** controls its own execution with system calls, as discussed under "Managing Applications" on page 4-2. Such an application may or may not obey the *msg_switches*, depending on how it was programmed.

The values used with the *msg_switches* (except **-plk** and **-noc**) are integer numbers of bytes. The default, maximum, and minimum values for these switches are described under "Default, Maximum, and Minimum Values" on page 8-20. The value you specify may be rounded up or down to ensure correct operation, as described under "Dependencies and Rounding" on page 8-21.

Summary of the Message-Passing Configuration Switches

The list of available *msg_switches* is as follows:

-plk Locks parts of the application into memory (see "Process Locking" on page 8-15 for more information).

-pkt *packet_size* Sets the size of each packet.

-mbf *message_buffer* Sets the total amount of memory allocated to message buffers in each process.

-noc *correspondents* Sets the total number of other processes from which each process expects to receive messages.

-mex *memory_export* Used in setting the maximum value for *memory_each*; otherwise ignored.

-mea *memory_each* Sets the amount of memory allocated to buffering messages from each correspondent.

-sth *send_threshold* Sets the threshold for sending multiple packets.

-sct *send_count* Sets the number of bytes to send right away when the available memory is above *send_threshold.*

-gth *give_threshold* Sets the threshold for "give me more messages" message.

See "Process Locking" on page 8-15, "Packetization" on page 8-16, and "System Message Buffers" on page 8-16 for more detailed information.

Default, Maximum, and Minimum Values

The default, maximum, and minimum values for the *msg_switches* are shown in Table 8-1.

Table 8-1. Message-Passing Configuration Switches

Switch	Parameter	Default	Maximum	Minimum
-plk	none	unlocked	n/a	n/a
-pkt	*packet_size*	8192 (for MP systems) or 1792 (for GP systems) or ((*memory_each* / 2) - **sizeof(xmsg_t)**[1]), whichever is less	8192 bytes (for MP systems) or 1792 bytes (for GP systems)	**sizeof(xmsg_t)**
-mbf	*message_buffer*	1MB + 128KB	32MB + (10 * *full_packet_size*[2])	(8 * **sizeof(xmsg_t)**) * (*correspondents* + 2) + (20 * **sizeof(xmsg_t)**)
-mex	*memory_export*	*message_buffer* - 128KB	*message_buffer* - 128KB	2 * (*correspondents* + 2) * (2 * *full_packet_size*)
-noc	*correspondents*	**numnodes()**	none	none
-mea	*memory_each*	(10 * *full_packet_size*) or maximum *memory_each*, whichever is less	1MB – 31 or (*memory_export* / 2) / (*correspondents* + 2), whichever is less	2 * *full_packet_size*
-sth	*send_threshold*	*memory_each* / 2	*memory_each* - 1	none
-sct	*send_count*	*memory_each* / 2	*memory_each*	*packet_size*
-gth	*give_threshold*	*packet_size*	*memory_each* / 2	*packet_size*

[1] **xmsg_t** is a type defined in *<mcmsg/mcmsg_xmsg.h>* that defines the message header sent along with each packet. The size of this type is currently 64 bytes.

[2] *full_packet_size* = *packet_size* + **sizeof(xmsg_t)**.

Dependencies and Rounding

As you can see from Table 8-1, the values for some of the *msg_switches* depend on the current values of other switches in a circular manner (for example, the default for *packet_size* depends on the value of *memory_each*, while the default for *memory_each* depends on the value of *packet_size*). These dependencies are resolved using the following procedure:

1. Set *packet_size*: If **-pkt** is specified, round the specified value up to a multiple of **sizeof(xmsg_t)**. Otherwise, use the default value.

2. Set *message_buffer*: If **-mbf** is specified, round the specified value up to a multiple of *full_packet_size*. Otherwise, use the default value.

3. Set *memory_export*: If **-mex** is specified, use the specified value. Otherwise, use the default value.

4. Set *memory_each*: If **-mea** is specified, round the specified value down to a multiple of **sizeof(xmsg_t)**. Otherwise, round the default value down to a multiple of **sizeof(xmsg_t)**.

5. Check that *memory_each* will hold at least two packets: If (*memory_each*/2) - **sizeof(xmsg_t)** is less than *packet_size*, reset *packet_size* to the value ((*memory_each*/2) - **sizeof(xmsg_t)**), round the resulting value down to a multiple of **sizeof(xmsg_t)**, then return to step 2. Otherwise, continue to step 6.

6. Set *send_threshold*: If **-sth** is specified, round the specified value down to a multiple of *packet_size*. Otherwise, round the default value down to a multiple of *packet_size*.

7. Set *send_count*: If **-sct** is specified, round the specified value down to a multiple of *packet_size*. Otherwise, round the default value down to a multiple of *packet_size*.

8. Set *give_threshold*: If **-gth** is specified, round the specified value down to a multiple of *packet_size*. Otherwise, round the default value down to a multiple of *packet_size*.

Recommendations

Because of the way message-passing flow control works, you should try to do all the following to achieve the best possible message-passing performance:

- Avoid paging, by keeping the application's memory requirements within available physical memory. Once you have done this, use the **-plk** switch if this is allowed at your site.

- Avoid blocking, by using asynchronous calls.

- Avoid system message buffering, by posting receives before the message is sent. (See "Avoid Message Buffering" on page 8-11 for tips on how to do this.)

It is important to make all three of these changes if possible. For example, even if you always post receives before the corresponding send occurs, system message buffering will still be necessary if the application buffer is paged out (or has never been paged in) when the message arrives.

If you cannot avoid system message buffering, you may be able to improve message-passing performance by increasing the *message_buffer* parameter (**-mbf**). This parameter determines the total amount of memory allocated to message buffers in each process; the other parameters determine how this memory is divided up. When you change the value of *message_buffer*, the defaults for the other parameters are automatically scaled to match the current *message_buffer* size. Increasing the *message_buffer* can increase the efficiency of message passing, but it also increases the memory usage of your application, which may cause paging and slow the application down. Once you have determined the optimal *message_buffer* size for your application, you can change the other parameters to fine-tune the usage of memory within the *message_buffer* and optimize message-passing performance.

The performance of some applications that use system message buffering can also be improved by reducing the *correspondents* parameter (**-noc**). This is particularly likely to help if your application slows down or hangs when you run it on more nodes. The **-noc** switch sets the "number of correspondents" for each process, which is the number of other processes from which the process receives messages. This number is used to determine how the memory allocated to buffering messages is divided up; more correspondents means that less memory is available for buffering messages from each correspondent. If you don't use **-noc**, the default for *correspondents* is **numnodes**(); that is, it is assumed that each process may receive messages from one process on each node. If you know that each process does not receive messages from every other node, using **-noc** to decrease the value of *correspondents* increases the *memory_each* buffer size, which can result in more efficient message passing (especially if the number of nodes is large). However, if the total number of other processes from which any process receives messages during the life of the application exceeds the value of *correspondents*, the application may run more slowly.

Note that certain global operations, such as sending to node -1 (which broadcasts a message to all nodes in the application) or calling **gdsum()**, can send messages to intermediate nodes. For example, sending to node -1 does not simply send one message to every other node; instead, it sends a message to several other nodes, which each send messages to several other nodes, and so on in a "message tree." This method is more efficient, but it means that if you use any global operations, the actual number of correspondents will be greater than the number of nodes from which each node receives explicit messages (by approximately the log of the number of nodes in the application).

I/O Performance

If your application performs I/O to files, you can use the following techniques to improve its I/O performance. Note: the term *request size* refers to the number of bytes specified in a single **read** or **write** operation. Techniques discussed in this section include:

- Use PFS file systems.
- Use **gopen**() instead of **open**().
- Use parallel I/O calls.
- Use asynchronous calls.
- Use the appropriate I/O mode.
- Align I/O buffers with virtual memory pages.
- Read or write whole file system blocks.
- Make good use of file striping.

See Chapter 5 for more information on these techniques.

Use PFS File Systems

Always store large data files in file systems of type *PFS* (*Parallel File System*). These file systems are optimized for large I/O requests (request sizes of 64K bytes or more) and simultaneous access by multiple nodes, and files in them can be larger than 2G bytes in size.

Use gopen() Instead of open()

If all nodes in an application open the same file, you should always use **gopen**() rather than **open**(). If all nodes call **open**(), each node sends an "open file" message to the same I/O node at the same time, which can swamp the I/O node with messages. But when all nodes call **gopen**(), only one node communicates with the I/O node; the open file descriptor is then broadcast to the other nodes in the application through efficient global communication techniques. If you *must* use **open**(), try to keep all the nodes from calling it at the same time (do *not* precede the **open**() with a **gsync**()).

Use Parallel I/O Calls

If you program in Fortran, you should always use the *parallel I/O calls*, such as **cread()**, to access your files. These calls give much better performance than the standard Fortran file I/O statements, such as **READ**.

If you program in C, you will not see any I/O performance increase from using parallel I/O calls, such as **cread()**, rather than standard UNIX I/O calls, such as **read()** (although **cread()** gives better performance than **fread()**, which is the C equivalent of Fortran's **READ**). However, you may be able to improve *computational* performance by using asynchronous I/O calls.

Use Asynchronous Calls

The parallel I/O calls are available in both *synchronous* versions (call names beginning with **c**) and *asynchronous* versions (call names beginning with **i**). Synchronous calls do not return until the I/O operation is complete; asynchronous calls return immediately, giving you an I/O ID that you can use to check when the operation is complete.

Although the synchronous calls are easier to use and have slightly lower overhead, you should use the asynchronous calls whenever the results of the call are not needed immediately. Using asynchronous calls can let your application do useful computation in the time when it would otherwise just be waiting for a large I/O operation to complete.

Use the Appropriate I/O Mode

When you use parallel I/O calls, you can choose from five I/O modes (**M_UNIX**, **M_LOG**, **M_SYNC**, **M_RECORD**, and **M_GLOBAL**), each of which is optimized for a particular pattern of file I/O. Be sure to use the correct I/O mode for your application's usage. In particular:

- Don't use **M_UNIX**, the default I/O mode, unless your application depends on its semantics.

- If all nodes read the same data from the same file at the same time, use **M_GLOBAL**.

- If all nodes read or write the same file, but each node is accessing a different part of the file, use **M_RECORD** if at all possible. This mode provides much higher multi-node performance than the other modes, all of which force reads and writes from different nodes to the same file to be performed in strict sequential order (this is required to preserve standard UNIX I/O semantics, but slows the application down).

- If **M_RECORD** cannot be used because the I/O request size is not constant across all compute nodes, use **M_SYNC** instead.

Align I/O Buffers with Virtual Memory Pages

Try to ensure that memory buffers used in I/O calls are aligned on an 8K-byte boundary whenever possible, to align with the operating system's virtual memory page size. This alignment is particularly important in scatter/gather operations with large request sizes to multiple I/O nodes. If you do not specify properly-aligned buffers, the software must copy the data to new, aligned buffers, which decreases performance.

To be sure the buffer is well-aligned, you should use **malloc()** to allocate it rather than allocating it statically. (This call is available in both Fortran and C.) Buffers allocated with the standard **malloc()** or its derivatives will always be aligned on a 32-byte boundary. See "Align Application Buffers" on page 8-12 for more information on aligning buffers.

Read or Write Whole File System Blocks

Disk space is allocated and managed in units called *file system blocks*. The size of each block in a file system is determined when the file system is created. For best performance, PFS file systems should have a file system block size of 64K bytes.

The file system block size is important because files always begin at a block boundary and data is most efficiently transferred to and from the physical disk in integer numbers of blocks. Furthermore, if a block is modified (but not entirely overwritten) by a write operation, the block may have to be read, modified in memory, and then written back.

Because of this, you will get the best I/O performance if each read or write request begins on a block boundary (a multiple of the block size from the beginning of the file) and the request size is a multiple of the file system block size.

To determine the block size of a file system, you can use the **statfs()** or **fstatfs()** call (see the *OSF/1 Programmer's Reference* for information on these calls), or ask your system administrator.

Make Good Use of File Striping

Files in PFS file systems are distributed, or *striped*, across several directories called *stripe directories*. The number of stripe directories in a PFS file system is called the *stripe factor*, and the amount of data from each file that is stored in each directory is called the *stripe unit*. The product of the stripe factor and the stripe unit is called the *full stripe size*. A PFS file system's stripe factor and stripe unit are set by the system administrator when the PFS file system is mounted.

Each stripe directory is typically on a separate disk, and each disk is typically controlled by a separate I/O node; you get the best I/O performance when you keep all the I/O nodes busy at once. You can use file striping to help you do this, with two different methods:

1. Use a request size equal to an integer multiple of the full stripe size, and make the starting address of each request the beginning of a full stripe. With this method, each I/O request goes to all the I/O nodes at once. This method can be used on any number of nodes.

2. Use a request size equal to the stripe unit size, make the starting address of each request the beginning of a stripe unit, and choose the starting address of each node's requests so that the nodes' requests are evenly distributed among the I/O nodes. With this method, each I/O request goes to just one I/O node, but the application's I/O requests are distributed among the I/O nodes. This method should be used only if the number of compute nodes is greater than or equal to the number of I/O nodes, preferably an integer multiple of the number of I/O nodes.

These two methods are illustrated in Figure 8-1. Note that method 1 uses fewer, larger requests and method 2 uses more, smaller requests. Method 1 is generally more efficient, but method 2 may give better performance for some situations (depending on the number of compute nodes, the number of I/O nodes, the amount of memory on each I/O node, and the size and frequency of requests). If possible, you should try both methods and use whichever is more efficient. The example program in */usr/share/examples/c/stripe* demonstrates the two methods, and you can use it to help you determine which method is best for your application. (Note that you will see more consistent results from one run to the next if the data size is large—8M bytes per node or more.)

You should always use the I/O mode **M_RECORD** when using these methods. **M_RECORD** is the most efficient I/O mode for this type of I/O, and automatically enforces the distribution of data among the I/O nodes. If you use **M_RECORD**, no file pointer calculation or seeking is required. For example:

```
fd = gopen(file, O_WRONLY|O_CREAT|O_TRUNC, M_RECORD, 0666);

while(data < end) {
    cwrite(fd, data, request_size);
    data += request_size;
}
```

Using this code, if *request_size* is equal to the full stripe size, each compute node automatically accesses all I/O nodes on each write (method 1). Alternatively, if *request_size* is equal to the stripe unit, each compute node automatically accesses exactly one I/O node on each write (method 2).

To determine the stripe factor and stripe unit of a PFS file system, you can use the **showfs** command (described under "Displaying File System Attributes" on page 5-5) or the **statpfs()** or **fstatpfs()** call (available only in C; described under "Getting Information About PFS File Systems" on page 5-41). The example program in */usr/share/examples/c/stripe* shows you how you can do this with **fstatpfs()**.

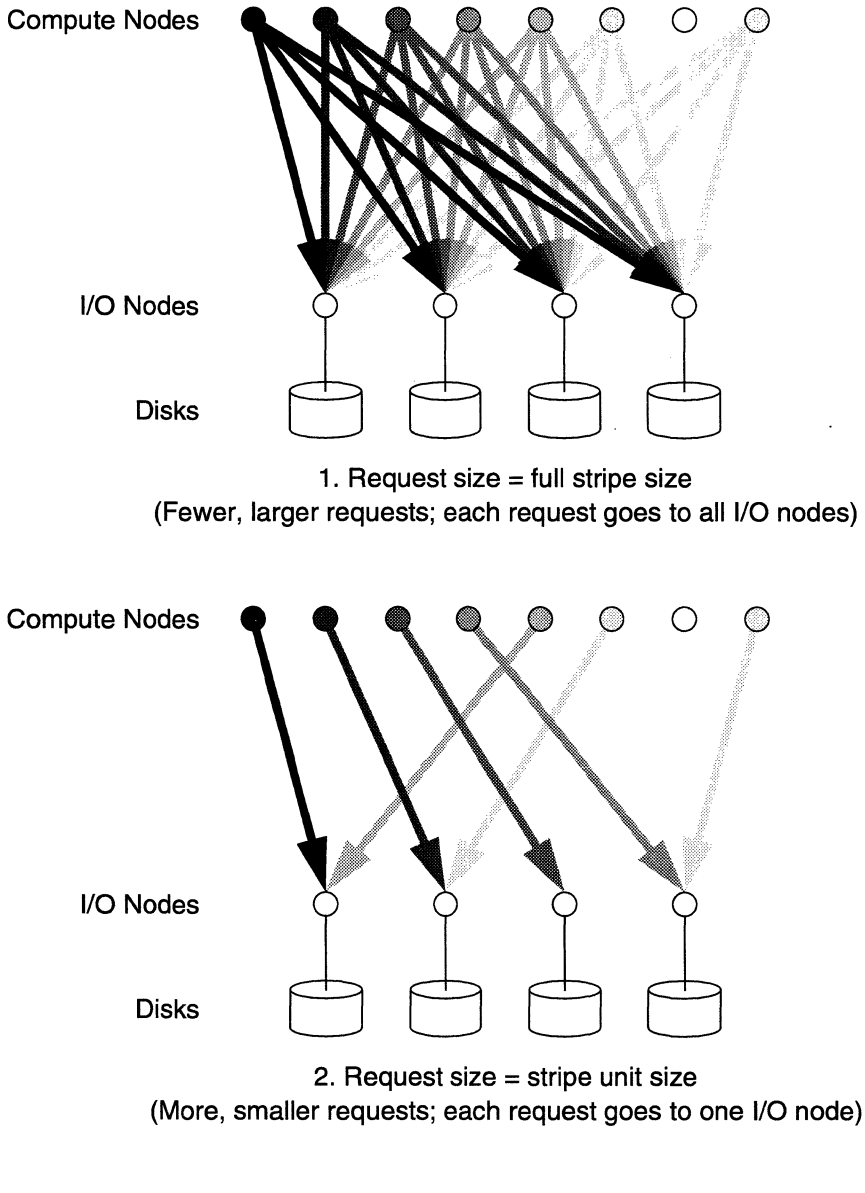

Compute Nodes
I/O Nodes
Disks
1. Request size = full stripe size
(Fewer, larger requests; each request goes to all I/O nodes)
Compute Nodes
I/O Nodes
Disks
2. Request size = stripe unit size
(More, smaller requests; each request goes to one I/O node)

Figure 8-1. Two Methods of Improving I/O Performance with M_RECORD

Summary of Commands and System Calls

A

This appendix summarizes the commands and system calls of the operating system. The complete syntax of each command and call is provided, along with a brief description of each. The C and Fortran versions of the calls are discussed in separate sections.

This appendix discusses only the commands and calls that are specific to the operating system. For information on the standard commands and calls of OSF/1, see the *OSF/1 Command Reference* and *OSF/1 Programmer's Reference.*

Command Summary

This section summarizes the commands discussed in Chapter 2 and Chapter 5. See the *Paragon*™ *System Commands Reference Manual* for more information on these commands.

Compiling and Linking Applications

Table A-1. Commands for Compiling and Linking Applications

Command Synopsis	Description
cc -nx [*switches*] *sourcefile...*	Compile an application written in C on a Paragon supercomputer.
f77 -nx [*switches*] *sourcefile...*	Compile an application written in Fortran on a Paragon supercomputer.
icc -nx [*switches*] *sourcefile...*	Compile an application written in C on a Paragon supercomputer or cross-development workstation.
if77 -nx [*switches*] *sourcefile...*	Compile an application written in Fortran on a Paragon supercomputer or cross-development workstation.

Running Applications

Table A-2. Commands for Running Applications

Command Synopsis	Description
application [**-sz** *size* | **-sz** *h***X***w* | **-nd** *h***X***w***:***n*] [**-rlx**] [**-pri** *priority*] [**-pt** *ptype*] [**-on** *nodespec*] [**-pn** *partition*] [**-nt** *nodetype*] [**-pkt** *packet_size*] [**-noc** *correspondents*] [**-mbf** *memory_buffer*] [**-mex** *memory_export*] [**-mea** *memory_each*] [**-sth** *send_threshold*] [**-sct** *send_count*] [**-gth** *give_threshold*] [**-plk**] [*application_args*] [**\;** *file* [**-pt** *ptype*] [**-on** *nodespec*] [*application_args*]] ...	Execute an application on a Paragon system.

Managing Partitions

Table A-3. Commands for Managing Partitions

Command Synopsis	Description
mkpart [**-sz** *size* | **-sz** *h***X***w* | **-nd** *nodespec*] [**-ss** | [[**-sps** | **-rq** *time*] [**-epl** *priority*]]] [**-mod** *mode*] [**-nt** *nodetype*] [**-rlx**] *partition*	Create a partition.
rmpart [**-f**] [**-r**] *partition*	Remove a partition.
showpart [**-f**] [**-l**] [**-p**] [**-nt** *nodetype*] [*partition*]	Show the characteristics of a partition.
lspart [**-r**] [**-l**] [**-p**] [*partition*]	List the subpartitions of a partition.
pspart [**-r**] [*partition*]	List the applications in a partition.
chpart [**-epl** *priority*] [**-g** *group*] [**-mod** *mode*] [**-nm** *name*] [**-o** *owner*[**.** *group*]] [**-rq** *time* | **-sps**] *partition*	Change certain partition characteristics.

Parallel File System Commands

Table A-4. Parallel File System Commands

Command Synopsis	Description
showfs [**-k**] [**-t** *type*] [*filesystem* I *directory*]	Display file system attributes.
lsize [**-a**] *size file* ...	Change the size of a file or files.
ls [-l] [-P] [*filesystem* I *directory*]	Lists and generates PFS information about files.

Miscellaneous Commands

Note: the commands shown in Table A-5 are not documented in this manual.

Table A-5. Miscellaneous Commands

Command Synopsis	Description
coreinfo [*corename*]	Displays summary information about a core file or the core files located in a core-file directory. (See the *Paragon*™ *System Commands Reference Manual* for more information.)
fsplit [*filename*]	Split one file containing several Fortran program units into several files containing one program unit each. (See the *Paragon*™ *System Commands Reference Manual* for more information.)
pmake [**-bcdeFikmnNpqrsStuUvw**] [**-C** *dir*] [**-f** *file*] [**-I** *dir*] [**-j** [*jobs*]] [**-l** [*load*]] [**-o** *file*] [**-P** *partition*] [**-W** *file*] [*macro_definition* ...] [*target* ...]	Parallel make utility that maintains up-to-date versions of target files and performs shell programs in parallel. (See the *Paragon*™ *System Application Tools User's Guide* for more information.)
sat [**-bchxV**] [**-d** *dir*] [**-l** *log*] [**-m** *mins*] [**-o** *output*] [**-p** *partition*] [**-r** *reps*] [*test* ...]	Run the Paragon system acceptance test. (See the *Paragon*™ *System Acceptance Test User's Guide* for more information.)

C System Call Summary

This section summarizes the C versions of the system calls discussed in Chapter 3, Chapter 4, Chapter 5, and Chapter 6. See the *Paragon™ System C Calls Reference Manual* for more information on these calls.

Process Characteristics

Table A-6. C Calls for Process Characteristics

Synopsis	Description
long **mynode**(void);	Obtain the calling process's node number.
long **numnodes**(void);	Obtain the number of nodes allocated to the current application.
long **myptype**(void);	Obtain the calling process's process type.
void **setptype**(long *ptype*);	Set the calling process's process type (only permitted if the process type is currently **INVALID_PTYPE**).
long **myhost**(void);	Obtain the controlling process's node number.

Synchronous Send and Receive

Table A-7. C Calls for Synchronous Send and Receive

Synopsis	Description
void **csend**(long *type*, char **buf*, long *count*, long *node*, long *ptype*);	Send a message, waiting for completion.
void **crecv**(long *typesel*, char **buf*, long *count*);	Receive a message, waiting for completion.
long **csendrecv**(long *type*, char **sbuf*, long *scount*, long *node*, long *ptype*, long *typesel*, char **rbuf*, long *rcount*);	Send a message and post a receive for the reply. Wait for completion.
void **gsendx**(long *type*, char **buf*, long *count*, long *nodes*[ ], long *nodecount*);	Send a message to a list of nodes, waiting for completion.

Asynchronous Send and Receive

Table A-8. C Calls for Asynchronous Send and Receive

Synopsis	Description
long **isend**(long *type*, char **buf*, long *count*, long *node*, long *ptype*);	Send a message without waiting for completion.
long **irecv**(long *typesel*, char **buf*, long *count*);	Receive a message without waiting for completion.
long **isendrecv**(long *type*, char **sbuf*, long *scount*, long *node*, long *ptype*, long *typesel*, char **rbuf*, long *rcount*);	Send a message and post a receive for the reply without waiting for completion.
long **msgdone**(long *mid*);	Determine whether a send or receive operation has completed.
void **msgwait**(long *mid*);	Wait for completion of a send or receive operation.
void **msgignore**(long *mid*);	Release a message ID as soon as a send or receive operation completes.
long **msgmerge**(long *mid1*, long *mid2*);	Merge two message IDs into a single ID that can be used to wait for completion of both operations.

Probing for Pending Messages

Table A-9. C Calls for Probing for Pending Messages

Synopsis	Description
void **cprobe**(long *typesel*);	Wait for a message of a selected type to arrive.
long **iprobe**(long *typesel*);	Determine whether a message of a selected type is pending.

Getting Information About Pending or Received Messages

Table A-10. C Calls for Getting Information About Pending or Received Messages

Synopsis	Description
long **infocount**(void);	Return size in bytes of a pending or received message.
long **infonode**(void);	Return node number of the node that sent a pending or received message.
long **infoptype**(void);	Return process type of the process that sent a pending or received message.
long **infotype**(void);	Return message type of a pending or received message.

Treating a Message as an Interrupt

Table A-11. C Calls for Treating a Message as an Interrupt

Synopsis	Description
void **hsend**(long *type*, char **buf*, long *count*, long *node*, long *ptype*, void (**handler*) ());	Send a message and set up a handler procedure to be called when the send completes.
void **hrecv**(long *typesel*, char **buf*, long *count*, void (**handler*) ());	Receive a message and set up a handler procedure to be called when the receive completes.
void **hsendrecv**(long *type*, char **sbuf*, long *scount*, long *node*, long *ptype*, long *typesel*, char **rbuf*, long *rcount*, void (**handler*) ());	Send a message and post a receive for the reply. Set up a handler procedure to be called when the reply arrives.
long **masktrap**(long *state*);	Enable or disable interrupts for message handlers. Required to prevent corruption of global variables.
void **hsendx**(long *type*, char **buf*, long *count*, long *node*, long *ptype*, void (**xhandler*) (), long *hparam*);	Send a message and set up an extended handler procedure to be called with the value *hparam* when the send completes. Allows handler sharing.

Extended Receive and Probe

Table A-12. C Calls for Extended Receive and Probe

Synopsis	Description
void **crecvx**(long *typesel*, char **buf*, long *count*, long *nodesel*, long *ptypesel*, long *info*[]);	Receive a message of a specified type from a specified sending node and process type, together with information about the message. Wait for completion.
long **irecvx**(long *typesel*, char **buf*, long *count*, long *nodesel*, long *ptypesel*, long *info*[]);	Receive a message of a specified type from a specified sending node and process type, together with information about the message. Do not wait for completion.
void **hrecvx**(long *typesel*, char **buf*, long *count*, long *nodesel*, long *ptypesel*, void (**xhandler*) (), long *hparam*);	Receive a message of a specified type from a specified sending node and process type. Set up an extended handler procedure to be called with information about the message and the value *hparam* when the receive completes.
void **cprobex**(long *typesel*, long *nodesel*, long *ptypesel*, long *info*[]);	Wait for a message of a specified type from a specified sending node and process type. Return information about the message.
long **iprobex**(long *typesel*, long *nodesel*, long *ptypesel*, long *info*[]);	Determine whether a message of a specified type from a specified sending node and process type is pending. If it is, return information about the message.

Global Operations

Table A-13. C Calls for Global Operations (1 of 3)

Synopsis	Description
void **gcol**(char *x*[ ], long *xlen*, char *y*[ ], long *ylen*, long **ncnt*);	Concatenation.
void **gcolx**(char *x*[ ], long *xlens*[ ], char *y*[ ]);	Concatenation for contributions of known length.
void **gdhigh**(double *x*[ ], long *n*, double *work*[ ]);	Vector double precision MAX.
void **gdlow**(double *x*[ ], long *n*, double *work*[ ]);	Vector double precision MIN.
void **gdprod**(double *x*[ ], long *n*, double *work*[ ]);	Vector double precision MULTIPLY.
void **gdsum**(double *x*[ ], long *n*, double *work*[ ]);	Vector double precision SUM.
void **giand**(long *x*[ ], long *n*, long *work*[ ]);	Vector integer bitwise AND.
void **gihigh**(long *x*[ ], long *n*, long *work*[ ]);	Vector integer MAX.

Table A-13. C Calls for Global Operations (2 of 3)

Synopsis	Description
void **gilow**(long *x*[ ], long *n*, long *work*[ ]);	Vector integer MIN.
void **gior**(long *x*[ ], long *n*, long *work*[ ]);	Vector integer bitwise OR.
void **giprod**(long *x*[ ], long *n*, long *work*[ ]);	Vector integer MULTIPLY.
void **gisum**(long *x*[ ], long *n*, long *work*[ ]);	Vector integer SUM.
void **gland**(long *x*[ ], long *n*, long *work*[ ]);	Vector logical AND.
void **glor**(long *x*[ ], long *n*, long *work*[ ]);	Vector logical inclusive OR.
void **gopf**(char *x*[ ], long *xlen*, char *work*[ ], long (**function*)());	Arbitrary commutative function.
void **gshigh**(float *x*[ ], long *n*, float *work*[ ]);	Vector real MAX.
void **gslow**(float *x*[ ], long *n*, float *work*[ ]);	Vector real MIN.

Table A-13. C Calls for Global Operations (3 of 3)

Synopsis	Description
void **gsprod**(float *x*[ ], long *n*, float *work*[ ]);	Vector real MULTIPLY.
void **gssum**(float *x*[ ], long *n*, float *work*[ ]);	Vector real SUM.
void **gsync**(void);	Global synchronization.

Controlling Application Execution

Table A-14. C Calls for Controlling Application Execution (1 of 2)

Synopsis	Description
long **nx_initve**(char **partition*, long *size*, char **account*, long **argc*, char **argv*[ ]);	Create a new application.
long **nx_initve_rect**(char **partition*, long *anchor*, long *rows*, long *cols*, char **account*, long **argc*, char **argv*[ ]);	Create a new application with a rectangular shape.
long **nx_initve_attr**(char **partition*, int **argc*, char **argv*[ ], [int *attribute*, { long | char * | long * } *value*,]... **NX_ATTR_END**);	Create a new application with specified attributes. (The type of each *value* parameter depends on the value of the previous *attribute* parameter.)
long **nx_pri**(long *pgroup*, long *priority*);	Set the priority of an application.

Table A-14. C Calls for Controlling Application Execution (2 of 2)

Synopsis	Description
long **nx_nfork**(long *node_list*[ ], long *numnodes*, long *ptype*, long *pid_list*[ ]);	Copy the current process onto some or all nodes of an application.
long **nx_load**(long *node_list*[ ], long *numnodes*, long *ptype*, long *pid_list*[ ], char **pathname*);	Execute a stored program on some or all nodes of an application.
long **nx_loadve**(long *node_list*[ ], long *numnodes*, long *ptype*, long *pid_list*[ ], char **pathname*, char **argv*[ ], char **envp*[ ]);	Execute a stored program on some or all nodes of an application, with specified argument list and environment.
long **nx_waitall**(void);	Wait for all application processes.

Getting Information About Applications

Table A-15. C Calls for Getting Information About Applications

Synopsis	Description
long **nx_app_rect**(long **rows*, long **cols*);	Obtain the height and width of the rectangle of nodes allocated to the current application.
int **nx_app_nodes**(pid_t *pgroup*, nx_nodes_t **node_list*, unsigned long **list_size*);	List the nodes allocated to an application.
int **nx_pspart**(char **partition*, nx_pspart_t ***pspart_list*, unsigned long **list_size*);	Obtain information about all applications and active subpartitions in a partition.

Partition Management

Table A-16. C Calls for Partition Management (1 of 2)

Synopsis	Description
long **nx_mkpart**(char **partition*, long *size*, long *type*);	Create a partition with a particular number of nodes.
long **nx_mkpart_rect**(char **partition*, long *rows*, long *cols*, long *type*);	Create a partition with a particular height and width.
long **nx_mkpart_map**(char **partition*, long *numnodes*, long *node_list*[ ], long *type*);	Create a partition with a specific set of nodes.
long **nx_mkpart_attr**(char **partition*, [int *attribute*, { long | char * | long * } *value*,]... **NX_ATTR_END**);	Create a partition with specified attributes. (The type of each *value* parameter depends on the value of the previous *attribute* parameter.)
long **nx_rmpart**(char **partition*, long*force*, long *recursive*);	Remove a partition.
int **nx_part_attr**(char **partition*, nx_part_info_t **attributes*);	Get a partition's attributes.
int **nx_part_nodes**(char **partition*, nx_nodes_t **node_list*, unsigned long **list_size*);	List the root node numbers for the nodes of a partition.
int **nx_node_attr**(char **partition*, char ****attributes*);	Get the node attributes for all nodes in a partition.
long **nx_chpart_name**(char **partition*, char **name*);	Change a partition's name.

Table A-16. C Calls for Partition Management (2 of 2)

Synopsis	Description
long **nx_chpart_mod**(char **partition*, long *mode*);	Change a partition's protection modes.
long **nx_chpart_epl**(char **partition*, long *priority*);	Change a partition's effective priority limit.
long **nx_chpart_rq**(char **partition*, long *rollin_quantum*);	Change a partition's rollin quantum.
long **nx_chpart_owner**(char **partition*, long *owner*, long *group*);	Change a partition's owner and group.
long **nx_chpart_sched**(char **partition*, int *sched_type*);	Change a partition's scheduling type.

Finding Unusable Nodes

Table A-17. C Calls for Finding Unusable Nodes

Synopsis	Description
int **nx_empty_nodes**(nx_nodes_t **node_list*, unsigned long **list_size*);	List the nodes that are empty slots.
int **nx_failed_nodes**(nx_nodes_t **node_list*, unsigned long **list_size*);	List the nodes that failed to boot.

Handling Errors

Table A-18. C Calls for Handling Errors

Synopsis	Description
_call(...);	Special version of *call* that returns error value to caller.
void **nx_perror**(char **string*);	Print an error message corresponding to the current value of *errno.*

Floating-Point Control

Table A-19. C Calls for Floating-Point Control

Synopsis	Description
int **isnan**(double *dsrc*);	Determine if a **double** value is Not-a-Number.
int **isnand**(double *dsrc*);	Determine if a **double** value is Not-a-Number.
int **isnanf**(float *fsrc*);	Determine if a **float** value is Not-a-Number.
fp_rnd **fpgetround**(void);	Get the floating-point rounding mode for the calling process.
fp_rnd **fpsetround**(fp_rnd *rnd_dir*);	Set the floating-point rounding mode for the calling process.
fp_except **fpgetmask**(void);	Get the floating-point exception mask for the calling process.
fp_except **fpsetmask**(fp_except *mask*);	Set the floating-point exception mask for the calling process.
fp_except **fpgetsticky**(void);	Get the floating-point exception sticky flags for the calling process.
fp_except **fpsetsticky**(fp_except *sticky*);	Set the floating-point exception sticky flags for the calling process.

Miscellaneous Calls

Table A-20. Miscellaneous C Calls

Synopsis	Description
void **flick**(void);	Temporarily relinquish the CPU to another process.
void **led**(long *state*);	Turn the node's green LED on or off.
double **dclock**(void);	Return time in seconds since booting the system.

iPSC® and Touchstone DELTA Compatibility

Table A-21. C Calls for iPSC® and Touchstone DELTA Compatibility (1 of 2)

Synopsis	Description
void **flushmsg**(long *typesel*, long *node*sel, long *ptypesel*);	Flush specified messages from the system.
long **ginv**(long *j*);	Return the position of an element in the binary-reflected gray code sequence. Inverse of **gray**().
long **gray**(long *j*);	Return the binary-reflected gray code for an integer.
void **hwclock**(esize_t **hwtime*);	Place the current value of the hardware counter into a 64-bit unsigned integer variable.
long **infopid**(void);	Return the process type of the process that sent a pending or received message.
void **killcube**(long *node*, long *ptype*);	Terminate and clear node process(es).
void **killproc**(long *node*, long *ptype*);	Terminate a node process.
void **load**(char **filename*, long *node*, long *ptype*);	Load a node process.

Table A-21. C Calls for iPSC® and Touchstone DELTA Compatibility (2 of 2)

Synopsis	Description
unsigned long **mclock**(void);	Return the time in milliseconds.
void **msgcancel**(long *mid*);	Cancel an asynchronous send or receive operation.
long **mypart**(long **rows*, long **cols*);	Obtain the height and width of the rectangle of nodes allocated to the current application.
long **mypid**(void);	Return the process type of the calling process.
long **nodedim**(void);	Return the dimension of the current application (the number of nodes allocated to the application is $2^{dimension}$).
long **restrictvol**(long *fildes*, long *nvol*, long *vollist*[]);	Return 0 (does nothing; provided for compatibility only).

I/O Modes

Table A-22. C Calls for I/O Modes

Synopsis	Description
int **gopen**(const char **path*, int *oflag*, int *iomode* [, mode_t *mode*]);	Open a file on all nodes and set its I/O mode.
void **setiomode**(int *fildes*, int *iomode*);	Set the I/O mode for a file.
long **iomode**(int *fildes*);	Return the current I/O mode for a file.

Reading and Writing Files in Parallel

Table A-23. C Calls for Reading and Writing Files in Parallel

Synopsis	Description
void **cread**(int *fildes*, char **buffer*, unsigned int *nbytes*);	Read from a file, waiting for completion.
void **cwrite**(int *fildes*, char **buffer*, unsigned int *nbytes*);	Write to a file, waiting for completion.
void **creadv**(int *fildes*, struct iovec **iov*, int *iovcount*);	Read from a file to irregularly-scattered buffers, waiting for completion.
void **cwritev**(int *fildes*, struct iovec **iov*, int *iovcount*);	Write to a file from irregularly-scattered buffers, waiting for completion.
long **iread**(int *fildes*, char **buffer*, unsigned int *nbytes*);	Read from a file without waiting for completion.
long **iwrite**(int *fildes*, char **buffer*, unsigned int *nbytes*);	Write to a file without waiting for completion.
long **ireadv**(int *fildes*, struct iovec **iov*, int *iovcount*);	Read from a file to irregularly-scattered buffers, without waiting for completion.
long **iwritev**(int *fildes*, struct iovec **iov*, int *iovcount*);	Write to a file from irregularly-scattered buffers, without waiting for completion.
long **iodone**(long *id*);	Determine whether an asynchronous I/O operation is complete. If complete, release the I/O ID.
void **iowait**(long *id*);	Wait for completion of an asynchronous I/O operation and release the I/O ID.

Detecting End-of-File and Moving the File Pointer

Table A-24. C Calls for Detecting End-of-File and Moving the File Pointer

Synopsis	Description
long **iseof**(int *fildes*);	Test for end-of-file.
off_t **lseek**(int *fildes*, off_t *offset*, int *whence*);	Move the read/write file pointer.

Increasing the Size of a File

Table A-25. C Calls for Increasing the Size of a File

Synopsis	Description
long **lsize**(int *fildes*, off_t *offset*, int *whence*);	Increase size of a file.

Extended File Manipulation

Table A-26. C Calls for Extended File Manipulation

Synopsis	Description
esize_t **eseek**(int *fildes*, esize_t *offset*, int *whence*);	Move file pointer in extended file.
esize_t **esize**(int *fildes*, esize_t *offset*, int *whence*);	Increase size of extended file.
long **estat**(char **path*, struct estat **buffer*);	Get status of extended file from pathname.
long **lestat**(char **path*, struct estat **buffer*);	Get status of extended file or symbolic link from pathname.
long **festat**(int *fildes*, struct estat **buffer*);	Get status of open extended file from file descriptor or unit.

Performing Extended Arithmetic

Table A-27. C Calls for Performing Extended Arithmetic

Synopsis	Description
esize_t **eadd**(esize_t *e1*, esize_t *e2*);	Add two extended integers.
long **ecmp**(esize_t *e1*, esize_t *e2*);	Compare two extended integers.
long **ediv**(esize_t *e*, long *n*);	Divide extended integer by integer.
long **emod**(esize_t *e*, long *n*);	Give extended integer modulo an integer (remainder when *e* is divided by *n*).
esize_t **emul**(esize_t *e*, long *n*);	Multiply extended integer by integer.
esize_t **esub**(esize_t *e1*, esize_t *e2*);	Subtract two extended integers.
void **etos**(esize_t *e*, char **s*);	Convert extended integer to string.
esize_t **stoe**(char **s*);	Convert string to extended integer.

Getting Information About PFS File Systems

Table A-28. C Calls for Getting Information About PFS™ File Systems

Synopsis	Description
long **getpfsinfo**(struct pfsmntinfo ***attrbufp*);	Get PFS-specific information about all mounted PFS file systems.
int **statpfs**(char **path*, struct estatfs **fs_buffer*, struct statpfs **pfs_buffer*, unsigned int *pfs_bufsize*);	Get PFS-specific and non-PFS-specific information for the file system containing *path.*
long **fstatpfs**(int *fildes*, struct estatfs **fs_buffer*, struct statpfs **pfs_buffer*, unsigned int *pfs_bufsize*);	Get PFS-specific and non-PFS-specific information for the file system containing the file open on *fildes*.

Managing Pthread Execution

Table A-29. C Calls for Managing Pthread Execution

Synopsis	Description
int **pthread_create**(pthread_t **thread*, pthread_attr_t *attr*, void *(**routine*)(void **arg*), void **arg*);	Creates a pthread.
pthread_t **pthread_self**(void);	Returns the ID of the calling pthread.
int **pthread_equal**(pthread_t *thread1*, pthread_t *thread2*);	Compares two pthread identifiers.
void **pthread_yield**(void);	Allows the scheduler to run another pthread instead of the current one.
void **pthread_exit**(void **status*);	Terminates the calling pthread.
int **pthread_join**(pthread_t *thread*, void ***status*);	Waits for a pthread to terminate.
int **pthread_detach**(pthread_t **thread*);	Detaches a pthread.

Managing Pthread Attributes

Table A-30. C Calls for Managing Pthread Attributes

Synopsis	Description
int **pthread_attr_create**(pthread_attr_t **attr*);	Creates a pthread attributes object.
int **pthread_attr_setstacksize**(pthread_attr_t **attr*, long *stacksize*);	Sets the value of the stack size attribute of a pthread attributes object.
int **pthread_attr_delete**(pthread_attr_t **attr*);	Deletes a pthread attributes object.
int **pthread_attr_getstacksize**(pthread_attr_t *attr*);	Returns the value of the stack size attribute of a pthread attributes object.

Managing Mutexes

Table A-31. C Calls for Managing Mutexes

Synopsis	Description
int **pthread_mutex_init**(pthread_mutex_t **mutex*, pthread_mutexattr_t *attr*);	Creates a mutex.
int **pthread_mutex_lock**(pthread_mutex_t **mutex*);	Locks a mutex.
int **pthread_mutex_trylock**(pthread_mutex_t **mutex*);	Tries once to lock a mutex.
int **pthread_mutex_unlock**(pthread_mutex_t **mutex*);	Unlocks a mutex.
int **pthread_mutex_destroy**(pthread_mutex_t **mutex*);	Deletes a mutex.
int **pthread_mutexattr_create**(pthread_mutexattr_t **attr*);	Creates a mutex attributes object.
int **pthread_mutexattr_delete**(pthread_mutexattr_t **attr*);	Deletes a mutex attributes object.

Using Condition Variables to Synchronize Pthreads

Table A-32. C Calls for Using Condition Variables to Synchronize Pthreads

Synopsis	Description
int **pthread_cond_init**(pthread_cond_t **cond*, pthread_condattr_t *attr*);	Creates a condition variable.
int **pthread_cond_wait**(pthread_cond_t **cond*, pthread_mutex_t **mutex*);	Waits on a condition variable.
int **pthread_cond_timedwait**(pthread_cond_t **cond*, pthread_mutex_t **mutex*, struct timespec **abstime*);	Waits on a condition variable for a specified period of time.
int **pthread_cond_signal**(pthread_cond_t **cond*);	Wakes up a pthread that is waiting on a condition variable.
int **pthread_cond_broadcast**(pthread_cond_t **cond*);	Wakes up all pthreads that are waiting on a condition variable.
int **pthread_cond_destroy**(pthread_cond_t **cond*);	Destroys a condition variable.
int **pthread_condattr_create**(pthread_condattr_t **attr*);	Creates a condition variable attributes object.
int **pthread_condattr_delete**(pthread_condattr_t **attr*);	Deletes a condition variable attributes object.

Canceling Pthreads

Table A-33. C Calls for Canceling Pthreads

Synopsis	Description
int **pthread_cancel**(pthread_t *thread*);	Requests cancellation of a pthread.
int **pthread_setcancel**(int *state*);	Enables or disables the general cancelability of the calling pthread.
int **pthread_setasynccancel**(int *state*);	Enables or disables the asynchronous cancelability of the calling pthread.
void **pthread_testcancel**(void);	Creates a cancellation point in the calling pthread.

Pthreads Cleanup Routines

Table A-34. C Calls for Pthreads Cleanup Routines

Synopsis	Description
void **pthread_cleanup_pop**(int *execute*);	Removes a routine from the top of the cleanup stack of the calling pthread and optionally executes it.
void **pthread_cleanup_push**(void (**routine*)(void **arg*), void **arg*);	Pushes a routine onto the cleanup stack of the calling pthread.

Managing Pthread Keys

Table A-35. C Calls for Managing Pthread Keys

Synopsis	Description
int **pthread_keycreate**(pthread_key_t **key*, void (**destructor*)(void **value*));	Creates a key to be used with pthread-specific data.
int **pthread_setspecific**(pthread_key_t *key*, void **value*);	Binds a pthread-specific value to a key.
int **pthread_getspecific**(pthread_key_t *key*, void ***value*);	Returns the value bound to a key.

Miscellaneous Pthread Calls

Table A-36. Miscellaneous Pthread Calls

Synopsis	Description
int **pthread_once**(pthread_once_t **once_block*, void(**routine*)());	Calls an initialization routine.
int **sigwait**(sigset_t **set*);	Suspends the calling pthread until one of a specified set of signals is received.

Fortran System Call Summary

This section summarizes the Fortran versions of the system calls discussed in Chapter 3, Chapter 4, and Chapter 5. See the *Paragon™ System Fortran Calls Reference Manual* for more information on these calls.

Process Characteristics

Table A-37. Fortran Calls for Process Characteristics

Synopsis	Description
INTEGER FUNCTION **MYNODE**()	Obtain the calling process's node number.
INTEGER FUNCTION **NUMNODES**()	Obtain the number of nodes allocated to the current application.
SUBROUTINE **SETPTYPE**(*ptype*) INTEGER *ptype*	Set the calling process's process type (only permitted if the process type is currently **INVALID_PTYPE**).
INTEGER FUNCTION **MYPTYPE**()	Obtain the calling process's process type.
INTEGER FUNCTION **MYHOST**()	Obtain the controlling process's node number.

Synchronous Send and Receive

Table A-38. Fortran Calls for Synchronous Send and Receive

Synopsis	Description
SUBROUTINE **CSEND**(*type*, *buf*, *count*, *node*, *ptype*) INTEGER *type* INTEGER *buf*(*) INTEGER *count* INTEGER *node* INTEGER *ptype*	Send a message, waiting for completion.
SUBROUTINE **CRECV**(*typesel*, *buf*, *count*) INTEGER *typesel* INTEGER *buf*(*) INTEGER *count*	Receive a message, waiting for completion.
INTEGER FUNCTION **CSENDRECV**(*type*, *sbuf*, *scount*, *node*, *ptype*, *typesel*, *rbuf*, *rcount*) INTEGER *type* INTEGER *sbuf*(*) INTEGER *scount* INTEGER *node* INTEGER *ptype* INTEGER *typesel* INTEGER *rbuf*(*) INTEGER *rcount*	Send a message and post a receive for the reply. Wait for completion.
SUBROUTINE **GSENDX**(*type*, *buf*, *count*, *nodes*, *nodecount*) INTEGER *type* INTEGER *buf*(*) INTEGER *count* INTEGER *nodes*(*) INTEGER *nodecount*	Send a message to a list of nodes, waiting for completion.

Asynchronous Send and Receive

Table A-39. Fortran Calls for Asynchronous Send and Receive (1 of 2)

Synopsis	Description
INTEGER FUNCTION **ISEND**(*type*, *buf*, *count*, *node*, *ptype*) INTEGER *type* INTEGER *buf*(*) INTEGER *count* INTEGER *node* INTEGER *ptype*	Send a message without waiting for completion.
INTEGER FUNCTION **IRECV**(*typesel*, *buf*, *count*) INTEGER *typesel* INTEGER *buf*(*) INTEGER *count*	Receive a message without waiting for completion.
INTEGER FUNCTION **ISENDRECV**(*type*, *sbuf*, *scount*, *node*, *ptype*, *typesel*, *rbuf*, *rcount*) INTEGER *type* INTEGER *sbuf*(*) INTEGER *scount* INTEGER *node* INTEGER *ptype* INTEGER *typesel* INTEGER *rbuf*(*) INTEGER *rcount*	Send a message and post a receive for the reply without waiting for completion.
INTEGER FUNCTION **MSGDONE**(*mid*) INTEGER *mid*	Determine whether a send or receive operation has completed.
SUBROUTINE **MSGWAIT**(*mid*) INTEGER *mid*	Wait for completion of a send or receive operation.

Table A-39. Fortran Calls for Asynchronous Send and Receive (2 of 2)

Synopsis	Description
SUBROUTINE **MSGIGNORE**(*mid*) INTEGER *mid*	Release a message ID as soon as a send or receive operation completes.
INTEGER FUNCTION **MSGMERGE**(*mid1*, *mid2*) INTEGER *mid1* INTEGER *mid2*	Merge two message IDs into a single ID that can be used to wait for completion of both operations.

Probing for Pending Messages

Table A-40. Fortran Calls for Probing for Pending Messages

Synopsis	Description
SUBROUTINE **CPROBE**(*typesel*) INTEGER *typesel*	Wait for a message of a selected type to arrive.
INTEGER FUNCTION **IPROBE**(*typesel*) INTEGER *typesel*	Determine whether a message of a selected type is pending.

Getting Information About Pending or Received Messages

Table A-41. Fortran Calls for Getting Information About Pending or Received Messages

Synopsis	Description
INTEGER FUNCTION **INFOCOUNT**()	Return size in bytes of a pending or received message.
INTEGER FUNCTION **INFONODE**()	Return node number of the node that sent a pending or received message.
INTEGER FUNCTION **INFOPTYPE**()	Return process type of the process that sent a pending or received message.
INTEGER FUNCTION **INFOTYPE**()	Return message type of a pending or received message.

Treating a Message as an Interrupt

Table A-42. Fortran Calls for Treating a Message as an Interrupt (1 of 2)

Synopsis	Description
SUBROUTINE **HSEND**(*type*, *buf*, *count*, *node*, *ptype*, *handler*) INTEGER *type* INTEGER *buf*(*) INTEGER *count* INTEGER *node* INTEGER *ptype* EXTERNAL *handler*	Send a message and set up a handler procedure to be called when the send completes.
SUBROUTINE **HRECV** (*typesel*, *buf*, *count*, *handler*) INTEGER *typesel* INTEGER *buf*(*) INTEGER *count* EXTERNAL *handler*	Receive a message and set up a handler procedure to be called when the receive completes.
SUBROUTINE **HSENDRECV**(*type*, *sbuf*, *scount*, *node*, *ptype*, *typesel*, *rbuf*, *rcount*, *handler*) INTEGER *type* INTEGER *sbuf*(*) INTEGER *scount* INTEGER *node* INTEGER *ptype* INTEGER *typesel* INTEGER *rbuf*(*) INTEGER *rcount* EXTERNAL *handler*	Send a message and post a receive for the reply. Set up a handler procedure to be called when the reply arrives.

Table A-42. Fortran Calls for Treating a Message as an Interrupt (2 of 2)

Synopsis	Description
INTEGER FUNCTION **MASKTRAP**(*state*) INTEGER *state*	Enable or disable interrupts for message handlers. Required to prevent corruption of global variables.
SUBROUTINE **HSENDX**(*type*, *buf*, *count*, *node*, *ptype*, *xhandler*, *hparam*) INTEGER *type* INTEGER *buf*(*) INTEGER *count* INTEGER *node* INTEGER *ptype* EXTERNAL *xhandler* INTEGER *hparam*	Send a message and set up an extended handler procedure to be called with the value *hparam* when the send completes. Allows handler sharing.

Extended Receive and Probe

Table A-43. Fortran Calls for Extended Receive and Probe (1 of 2)

Synopsis	Description
SUBROUTINE **CRECVX**(*typesel*, *buf*, *count*, *nodesel*, *ptypesel*, *info*) INTEGER *typesel* INTEGER *buf*(*) INTEGER *count* INTEGER *nodesel* INTEGER *ptypesel* INTEGER *info*(8)	Receive a message of a specified type from a specified sending node and process type, together with information about the message. Wait for completion.
INTEGER FUNCTION **IRECVX**(*typesel*, *buf*, *count*, *nodesel*, *ptypesel*, *info*) INTEGER *typesel* INTEGER *buf*(*) INTEGER *count* INTEGER *nodesel* INTEGER *ptypesel* INTEGER *info*(8)	Receive a message of a specified type from a specified sending node and process type, together with information about the message. Do not wait for completion.

Table A-43. Fortran Calls for Extended Receive and Probe (2 of 2)

Synopsis	Description
SUBROUTINE **HRECVX**(*typesel*, *buf*, *count*, *nodesel*, *ptypesel*, *xhandler*, *hparam*) INTEGER *typesel* INTEGER *buf*(*) INTEGER *count* INTEGER *nodesel* INTEGER *ptypesel* EXTERNAL *xhandler* INTEGER *hparam*	Receive a message of a specified type from a specified sending node and process type. Set up an extended handler procedure to be called with information about the message and the value *hparam* when the receive completes.
SUBROUTINE **CPROBEX**(*typesel*, *nodesel*, *ptypesel*, *info*) INTEGER *typesel* INTEGER *nodesel* INTEGER *ptypesel* INTEGER *info*(8)	Wait for a message of a specified type from a specified sending node and process type. Return information about the message.
INTEGER FUNCTION **IPROBEX**(*typesel*, *nodesel*, *ptypesel*, *info*) INTEGER *typesel* INTEGER *nodesel* INTEGER *ptypesel* INTEGER *info*(8)	Determine whether a message of a specified type from a specified sending node and process type is pending. If it is, return information about the message.

Global Operations

Table A-44. Fortran Calls for Global Operations (1 of 3)

Synopsis	Description
SUBROUTINE **GCOL**(*x*, *xlen*, *y*, *ylen*, *ncnt*) INTEGER *x*(*) INTEGER *xlen* INTEGER *y*(*) INTEGER *ylen* INTEGER *ncnt*	Concatenation.
SUBROUTINE **GCOLX**(*x*, *xlens*, *y*) INTEGER *x*(*) INTEGER *xlens*(*) INTEGER *y*(*)	Concatenation for contributions of known length.
SUBROUTINE **GDHIGH**(*x*, *n*, *work*) DOUBLE PRECISION *x*(*) INTEGER *n* DOUBLE PRECISION *work*(*)	Vector double precision MAX.
SUBROUTINE **GDLOW**(*x*, *n*, *work*) DOUBLE PRECISION *x*(*) INTEGER *n* DOUBLE PRECISION *work*(*)	Vector double precision MIN.
SUBROUTINE **GDPROD**(*x*, *n*, *work*) DOUBLE PRECISION *x*(*) INTEGER *n* DOUBLE PRECISION *work*(*)	Vector double precision MULTIPLY.
SUBROUTINE **GDSUM**(*x*, *n*, *work*) DOUBLE PRECISION *x*(*) INTEGER *n* DOUBLE PRECISION *work*(*)	Vector double precision SUM.
SUBROUTINE **GIAND**(*x*, *n*, *work*) INTEGER *x*(*) INTEGER *n* INTEGER *work*(*)	Vector integer bitwise AND.

Table A-44. Fortran Calls for Global Operations (2 of 3)

Synopsis	Description
SUBROUTINE **GIHIGH**(*x*, *n*, *work*) INTEGER *x*(*) INTEGER *n* INTEGER *work*(*)	Vector integer MAX.
SUBROUTINE **GILOW**(*x*, *n*, *work*) INTEGER *x*(*) INTEGER *n* INTEGER *work*(*)	Vector integer MIN.
SUBROUTINE **GIOR**(*x*, *n*, *work*) INTEGER *x*(*) INTEGER *n* INTEGER *work*(*)	Vector integer bitwise OR.
SUBROUTINE **GIPROD**(*x*, *n*, *work*) INTEGER *x*(*) INTEGER *n* INTEGER *work*(*)	Vector integer MULTIPLY.
SUBROUTINE **GISUM**(*x*, *n*, *work*) INTEGER *x*(*) INTEGER *n* INTEGER *work*(*)	Vector integer SUM.
SUBROUTINE **GLAND**(*x*, *n*, *work*) LOGICAL *x*(*) INTEGER *n* LOGICAL *work*(*)	Vector logical AND.
SUBROUTINE **GLOR**(*x*, *n*, *work*) LOGICAL *x*(*) INTEGER *n* LOGICAL *work*(*)	Vector logical inclusive OR.

Table A-44. Fortran Calls for Global Operations (3 of 3)

Synopsis	Description
SUBROUTINE **GOPF**(*x*, *xlen*, *work*, *function*) INTEGER *x*(*) INTEGER *xlen* INTEGER *work*(*) EXTERNAL *function*	Arbitrary commutative function.
SUBROUTINE **GSHIGH**(*x*, *n*, *work*) REAL *x*(*) INTEGER *n* REAL *work*(*)	Vector real MAX.
SUBROUTINE **GSLOW**(*x*, *n*, *work*) REAL *x*(*) INTEGER *n* REAL *work*(*)	Vector real MIN.
SUBROUTINE **GSPROD**(*x*, *n*, *work*) REAL *x*(*) INTEGER *n* REAL *work*(*)	Vector real MULTIPLY.
SUBROUTINE **GSSUM**(*x*, *n*, *work*) REAL *x*(*) INTEGER *n* REAL *work*(*)	Vector real SUM.
SUBROUTINE **GSYNC**()	Global synchronization.

Controlling Application Execution

Table A-45. Fortran Calls for Controlling Application Execution (1 of 2)

Synopsis	Description
INTEGER FUNCTION **NX_INITVE**(*partition*, *size*, *account*, *argc*, *argv*) CHARACTER partition*(*) INTEGER size CHARACTER account*(*) INTEGER argc INTEGER argv	Create a new application.
INTEGER FUNCTION **NX_INITVE_RECT**(*partition*, *anchor*, *rows*, *cols*, *account*, *argc*, *argv*) CHARACTER partition*(*) INTEGER anchor INTEGER rows INTEGER cols CHARACTER account*(*) INTEGER argc INTEGER argv	Create a new application with a rectangular shape.
INTEGER FUNCTION **NX_INITVE_ATTR**(*partition*, *argc*, *argv*, [*attribute*, *value*,]... **NX_ATTR_END**) CHARACTER partition*(*) INTEGER argc INTEGER argv INTEGER attribute { INTEGER value | CHARACTER value*(*) | INTEGER value(2) }	Create a new application with specified attributes. (The type of each *value* parameter depends on the value of the previous *attribute* parameter.)
INTEGER FUNCTION **NX_PRI**(*pgroup*, *priority*) INTEGER pgroup INTEGER priority	Set the priority of an application.

Table A-45. Fortran Calls for Controlling Application Execution (2 of 2)

Synopsis	Description
INTEGER FUNCTION **NX_NFORK**(*node_list*, *numnodes*, *ptype*, *pid_list*) INTEGER *node_list*(*) INTEGER *numnodes* INTEGER *ptype* INTEGER *pid_list*(*)	Copy the current process onto some or all nodes of an application.
INTEGER FUNCTION **NX_LOAD**(*node_list*, *numnodes*, *ptype*, *pid_list*, *pathname*) INTEGER *node_list*(*) INTEGER *numnodes* INTEGER *ptype* INTEGER *pid_list*(*) CHARACTER *pathname**(*)	Execute a stored program on some or all nodes of an application.
INTEGER FUNCTION **NX_LOADVE**(*node_list*, *numnodes*, *ptype*, *pid_list*, *pathname*, *argv*, *envp*) INTEGER *node_list*(*) INTEGER *numnodes* INTEGER *ptype* INTEGER *pid_list*(*) CHARACTER *pathname**(*) INTEGER *argv* INTEGER *envp*	Execute a stored program on some or all nodes of an application, with specified argument list and environment.
SUBROUTINE **NX_WAITALL**()	Wait for all application processes.

Getting Information About Applications

Table A-46. Fortran Calls for Getting Information About Applications

Synopsis	Description
INTEGER FUNCTION **NX_APP_RECT**(*rows*, *cols*) INTEGER *rows* INTEGER *cols*	Obtain the height and width of the rectangle of nodes allocated to the current application.
INTEGER FUNCTION **NX_APP_NODES**(*pgroup*, *ptr*, *list_size*) INTEGER *pgroup* POINTER (*ptr*, *node_list*(1)) INTEGER *list_size*	List the nodes allocated to an application.

Partition Management

Table A-47. Fortran Calls for Partition Management (1 of 3)

Synopsis	Description
INTEGER FUNCTION **NX_MKPART**(*partition*, *size*, *type*) CHARACTER *partition**(*) INTEGER *size* INTEGER *type*	Create a partition with a particular number of nodes.
INTEGER FUNCTION **NX_MKPART_RECT**(*partition*, *rows*, *cols*, *type*) CHARACTER *partition**(*) INTEGER *rows* INTEGER *cols* INTEGER *type*	Create a partition with a particular height and width.

Table A-47. Fortran Calls for Partition Management (2 of 3)

Synopsis	Description
INTEGER FUNCTION **NX_MKPART_MAP**(partition, numnodes, node_list, type) CHARACTER partition*(*) INTEGER numnodes INTEGER node_list(*) INTEGER type	Create a partition with a specific set of nodes.
INTEGER FUNCTION **NX_MKPART_ATTR**(partition, [attribute, value,]... **NX_ATTR_END**) CHARACTER partition*(*) INTEGER attribute { INTEGER value | CHARACTER value*(*) | INTEGER value(2) }	Create a partition with specified attributes. (The type of each value parameter depends on the value of the previous attribute parameter.)
INTEGER FUNCTION **NX_RMPART**(pathname,force, recursive) CHARACTER partition*(*) INTEGER force INTEGER recursive	Remove a partition.
INTEGER FUNCTION **NX_PART_ATTR**(partition, attributes) CHARACTER partition*(*) RECORD /nx_part_info_t/ attributes	Get a partition's attributes.
INTEGER FUNCTION **NX_PART_NODES**(partition, ptr, list_size) CHARACTER partition*(*) POINTER (ptr, node_list(1)) INTEGER list_size	List the root node numbers for the nodes of a partition.
INTEGER FUNCTION **NX_CHPART_NAME**(partition, name) CHARACTER partition*(*) CHARACTER name*(*)	Change a partition's name.

Table A-47. Fortran Calls for Partition Management (3 of 3)

Synopsis	Description
INTEGER FUNCTION **NX_CHPART_MOD**(*partition*, *mode*) CHARACTER *partition**(*) INTEGER *mode*	Change a partition's protection modes.
INTEGER FUNCTION **NX_CHPART_EPL**(*partition*, *priority*) CHARACTER *partition**(*) INTEGER *priority*	Change a partition's effective priority limit.
INTEGER FUNCTION **NX_CHPART_RQ**(*partition*, *rollin_quantum*) CHARACTER *partition**(*) INTEGER *rollin_quantum*	Change a partition's rollin quantum.
INTEGER FUNCTION **NX_CHPART_OWNER**(*partition*, *owner*, *group*) CHARACTER *partition**(*) INTEGER *owner* INTEGER *group*	Change a partition's owner and group.
INTEGER FUNCTION **NX_CHPART_SCHED**(*partition*, *sched_type*) CHARACTER *partition**(*) INTEGER *sched_type*	Change a partition's rollin quantum.

Finding Unusable Nodes

Table A-48. Fortran Calls for Finding Unusable Nodes

Synopsis	Description
INTEGER FUNCTION **NX_EMPTY_NODES**(*ptr*, *list_size*) POINTER (*ptr*, *node_list*(1)) INTEGER *list_size*	List the nodes that are empty slots.
INTEGER FUNCTION **NX_FAILED_NODES**(*ptr*, *list_size*) POINTER (*ptr*, *node_list*(1)) INTEGER *list_size*	List the nodes that failed to boot.

Handling Errors

Table A-49. Fortran Calls for Handling Errors

Synopsis	Description
SUBROUTINE **NX_PERROR**(*string*) CHARACTER *string**(*)	Print an error message corresponding to the current value of *errno*.

Floating-Point Control

Table A-50. Fortran Calls for Floating-Point Control

Synopsis	Description
INTEGER FUNCTION **FPSETMASK**(*mask*) INTEGER *mask*	Set the floating-point exception mask for the calling process.

Miscellaneous Calls

Table A-51. Miscellaneous Fortran Calls

Synopsis	Description
SUBROUTINE **FLICK**()	Temporarily relinquish the CPU to another process.
SUBROUTINE **LED**(*state*) INTEGER *state*	Turn the node's green LED on or off.
DOUBLE PRECISION FUNCTION **DCLOCK**()	Return time in seconds since booting the system.

iPSC® and Touchstone DELTA Compatibility

Table A-52. Fortran Calls for iPSC® and Touchstone DELTA Compatibility (1 of 2)

Synopsis	Description
SUBROUTINE **FLUSHMSG**(*typesel*, *node*sel, *ptypesel*) INTEGER *typesel* INTEGER *node*sel INTEGER *ptypesel*	Flush specified messages from the system.
INTEGER FUNCTION **GINV**(*graycode*) INTEGER *graycode*	Return the position of an element in the binary-reflected gray code sequence. Inverse of **gray**().
INTEGER FUNCTION **GRAY**(*position*) INTEGER *position*	Return the binary-reflected gray code for an integer.
SUBROUTINE **HWCLOCK**(*hwtime*) INTEGER *hwtime*(2)	Place the current value of the hardware counter into a 64-bit unsigned integer variable.
INTEGER FUNCTION **INFOPID**()	Return the process type of the process that sent a pending or received message.
SUBROUTINE **KILLCUBE**(*node*, *pid*) INTEGER *node* INTEGER *pid*	Terminate and clear node process(es).

Table A-52. Fortran Calls for iPSC® and Touchstone DELTA Compatibility (2 of 2)

Synopsis	Description
SUBROUTINE **KILLPROC**(*node*, *pid*) INTEGER *node* INTEGER *pid*	Terminate a node process.
SUBROUTINE **LOAD**(*filename*, *node*, *pid*) CHARACTER *filename**(*) INTEGER *node* INTEGER *pid*	Load a node process.
INTEGER FUNCTION **MCLOCK**()	Return the time in milliseconds.
SUBROUTINE **MSGCANCEL**(*mid*) INTEGER *mid*	Cancel an asynchronous send or receive operation.
INTEGER FUNCTION **MYPART**(*rows*, *cols*) INTEGER *rows* INTEGER *cols*	Obtain the height and width of the rectangle of nodes allocated to the current application.
INTEGER FUNCTION **MYPID**()	Return the process type of the calling process.
INTEGER FUNCTION **NODEDIM**()	Return the dimension of the current application (the number of nodes allocated to the application is $2^{dimension}$).
INTEGER FUNCTION **RESTRICTVOL**(*unit*, *nvol*, *vollist*) INTEGER *unit* INTEGER *nvol* INTEGER *vollist*(*)	Return 0 (does nothing; provided for compatibility only).

I/O Modes

Table A-53. Fortran Calls for I/O Modes

Synopsis	Description
SUBROUTINE **GOPEN**(*unit*, *path*, *iomode*) INTEGER *unit* CHARACTER *path**(*) INTEGER *iomode*	Open a file on all nodes and set its I/O mode.
SUBROUTINE **SETIOMODE**(*unit*, *iomode*) INTEGER *unit* INTEGER *iomode*	Set the I/O mode for a file.
INTEGER FUNCTION **IOMODE**(*unit*) INTEGER *unit*	Return the current I/O mode for a file.

Reading and Writing Files in Parallel

Table A-54. Fortran Calls for Reading and Writing Files in Parallel (1 of 2)

Synopsis	Description
SUBROUTINE **CREAD**(*unit*, *buffer*, *nbytes*) INTEGER *unit* INTEGER *buffer*(*) INTEGER *nbytes*	Read from a file, waiting for completion.
SUBROUTINE **CWRITE**(*unit*, *buffer*, *nbytes*) INTEGER *unit* INTEGER *buffer*(*) INTEGER *nbytes*	Write to a file, waiting for completion.
SUBROUTINE **CREADV**(*unit*, *iov*, *iovcnt*) INTEGER *unit* INTEGER *iov*(*) INTEGER *iovcnt*	Read from a file to irregularly-scattered buffers, waiting for completion.

Table A-54. Fortran Calls for Reading and Writing Files in Parallel (2 of 2)

Synopsis	Description
SUBROUTINE **CWRITEV**(*unit*, *iov*, *iovcnt*) INTEGER *unit* INTEGER *iov*(*) INTEGER *iovcnt*	Write to a file from irregularly-scattered buffers, waiting for completion.
INTEGER FUNCTION **IREAD**(*unit*, *buffer*, *nbytes*) INTEGER *unit* INTEGER *buffer*(*) INTEGER *nbytes*	Read from a file without waiting for completion.
INTEGER FUNCTION **IWRITE**(*unit*, *buffer*, *nbytes*) INTEGER *unit* INTEGER *buffer*(*) INTEGER *nbytes*	Write to a file, waiting for completion.
INTEGER FUNCTION **IREADV**(*unit*, *iov*, *iovcnt*) INTEGER *unit* INTEGER *iov*(*) INTEGER *iovcnt*	Read from a file to irregularly-scattered buffers, without waiting for completion.
INTEGER FUNCTION **IWRITEV**(*unit*, *iov*, *iovcnt*) INTEGER *unit* INTEGER *iov*(*) INTEGER *iovcnt*	Write to a file from irregularly-scattered buffers, without waiting for completion.
INTEGER FUNCTION **IODONE**(*id*) INTEGER *id*	Determine whether an asynchronous I/O operation is complete. If complete, release the I/O ID.
SUBROUTINE **IOWAIT**(*id*) INTEGER *id*	Wait for completion of an asynchronous I/O operation and release the I/O ID.

Detecting End-of-File and Moving the File Pointer

Table A-55. Fortran Calls for Detecting End-of-File and Moving the File Pointer

Synopsis	Description
INTEGER FUNCTION **ISEOF**(*unit*) INTEGER *unit*	Test for end-of-file.
INTEGER FUNCTION **LSEEK**(*unit*, *offset*, *whence*) INTEGER *unit* INTEGER *offset* INTEGER *whence*	Move the read/write file pointer.

Flushing Fortran Buffered I/O

Table A-56. Fortran Calls for Flushing Buffered I/O

Synopsis	Description
SUBROUTINE **FORCEFLUSH**()	Cause all buffered I/O to be flushed if an exception occurs.
SUBROUTINE **FORFLUSH**(*unit*) INTEGER *unit*	Flush all buffered I/O on a particular unit.

Increasing the Size of a File

Table A-57. Fortran Calls for Increasing the Size of a File

Synopsis	Description
INTEGER FUNCTION **LSIZE**(*unit*, *offset*, *whence*) INTEGER *unit* INTEGER *offset* INTEGER *whence*	Increase size of a file.

Extended File Manipulation

Table A-58. Fortran Calls for Extended File Manipulation

Synopsis	Description
SUBROUTINE **ESEEK**(*unit*, *offset*, *whence*, *newpos*) INTEGER *unit* INTEGER *offset*(2) INTEGER *whence* INTEGER *newpos*(2)	Move file pointer in extended file.
SUBROUTINE **ESIZE**(*unit*, *offset*, *whence*, *newsize*) INTEGER *unit* INTEGER *offset*(2) INTEGER *whence* INTEGER *newsize*(2)	Increase size of extended file.

Performing Extended Arithmetic

Table A-59. Fortran Calls for Performing Extended Arithmetic

Synopsis	Description
SUBROUTINE **EADD**(*e1*, *e2*, *eresult*) INTEGER *e1*(2) INTEGER *e2*(2) INTEGER *eresult*(2)	Add two extended integers.
INTEGER FUNCTION **ECMP**(*e1*, *e2*) INTEGER *e1*(2) INTEGER *e2*(2)	Compare two extended integers.
SUBROUTINE **EDIV**(*e*, *n*, *result*) INTEGER *e*(2) INTEGER *n* INTEGER *result*	Divide extended integer by integer.
SUBROUTINE **EMOD**(*e*, *n*, *result*) INTEGER *e*(2) INTEGER *n* INTEGER *result*	Give extended integer modulo an integer (remainder when *e* is divided by *n*).
SUBROUTINE **EMUL**(*e*, *n*, *eresult*) INTEGER *e*(2) INTEGER *n* INTEGER *eresult*(2)	Multiply extended integer by integer.
SUBROUTINE **ESUB**(*e1*, *e2*, *eresult*) INTEGER *e1*(2) INTEGER *e2*(2) INTEGER *eresult*(2)	Subtract two extended integers.
SUBROUTINE **ETOS**(*e*, *s*) INTEGER *e*(2) CHARACTER *s*(*)	Convert extended integer to string.
SUBROUTINE **STOE**(*s*, *e*) CHARACTER *s*(*) INTEGER *e*(2)	Convert string to extended integer.

iPSC® System Compatibility B

Introduction

This appendix gives you information you can use to port programs to the Paragon system from the iPSC® series of supercomputers from Intel Scalable Systems Division.

This appendix lists the differences between iPSC system commands and system calls and those of the operating system, and suggests alternatives that you can use for commands and calls that are not supported. Commands and calls that are not listed here should work the same in the operating system as they do in the iPSC system.

General Compatibility Issues

In general, iPSC system programs can simply be recompiled and executed on the Paragon system. However, keep in mind the following basic differences between the two systems:

- There is no SRM. The Diagnostic Station is used only for system administration; all software development is done either on remote workstations or on the Paragon system itself. Parallel applications are run only on the Paragon system.

- Host programs are not directly supported. See "Host Calls" on page B-9 for more information.

- The node network is a 2-D mesh rather than a hypercube. You might want to change the data distribution in your application to take advantage of the different system topology.

- An application can run on *any* number of nodes from 1 to the size of the compute partition (up to several thousand nodes). If your application depends on the number of nodes being a power of two or no greater than 128, you should re-write it so that it works on any number of nodes. If this is not possible, you should have the application print an error message if **numnodes()** is not a power of two or is too large for the application to handle.

- If a message arrives at a node before the receive for the message has been posted, the message is stored in a system buffer. In the iPSC system, the space available for these system buffers is the entire free physical memory of the node. In the Paragon system, this space is more limited (1M bytes by default). This limitation results from the fact that the Paragon system supports multiple processes per node.

NOTE

Because of this limitation, iPSC system applications that use large amounts of system message buffering may slow down or hang on the Paragon system, especially when run on large numbers of nodes.

If this occurs, you can increase the system message buffering space with the **-mbf** switch, as described under "System Message Buffers" on page 8-16. However, it would be better to re-write the application so that receives are always posted before the message arrives, as discussed under "Avoid Message Buffering" on page 8-11.

- The term *process ID*, or *PID*, is used differently. In the iPSC system, each process has a *UNIX PID* used by the OS and an *NX PID* used for message passing. In the Paragon system, the "UNIX PID" is just called the *PID*, and the "NX PID" is called the *process type* or *ptype*. Although the names have changed, the software works the same. For example, **mypid()** and **infopid()** are supported as equivalents to **myptype()** and **infoptype()**. Exception: on the iPSC/860, the NX PID is always 0; in the Paragon system, the process type can be any integer from 0 to 2,147,483,647 ($2^{31} - 1$) inclusive (but is usually 0).

- Force types (special message types that use a limited flow control technique) are fully supported and work the same as they do in the iPSC system. However, in the Paragon system regular messages are just as fast as force type messages, so force types are not needed for performance.

New Features

The operating system offers the following features that were not available on the iPSC system. You can use these features to improve the performance and readability of your programs.

- You can use the complete set of OSF/1 commands on the Paragon system, as discussed in Chapter 2.

- You can execute an application on multiple nodes just by typing its name on the command line, using command-line switches to control its execution, as discussed under "Running Applications" on page 2-11.

- You can control the values of some important message-passing configuration parameters, as discussed under "Message-Passing Configuration Switches" on page 8-18.

- You can allocate groups of nodes of any size and shape, and control the scheduling characteristics of applications that run in them, as discussed under "Managing Partitions" on page 2-30 and "Managing Partitions" on page 4-32.

- You can have more than one process per node, as discussed under "Process Characteristics" on page 3-3. When sending messages, you specify a process by its *process type* (equivalent to the "NX PID" in the iPSC system).

- You can tell the system to discard an asynchronous message ID as soon as the send or receive completes with **msgignore()**, as discussed under "Asynchronous Send and Receive" on page 3-10.

- You can merge together a number of asynchronous message-passing requests and wait for all of them to complete in a single call with **msgmerge()**, as discussed under "Merging Message IDs" on page 3-13.

- Global sends use a -1 value for the node parameter in a send system call. Global sends are now implemented using a tree-structured store-and-receive message passing strategy. As each intermediate node receives a message, it forwards the message to other nodes. If a node fails to receive a message from a global send, other nodes that depend on that node to receive the message will never receive the message. This can cause an application to hang.

- The **hsend()** and **hrecv()** calls now run concurrently with the main program rather than interrupting it. You still need to protect critical sections of code, as discussed under "Preventing Interrupts" on page 3-22.

- You can pass a parameter to a message interrupt handler with **hsendx()**, as discussed under "Treating a Message as an Interrupt" on page 3-18.

- You can receive or probe for a message based on its sender, and receive information about a message along with the message, with the **...x()** calls, as discussed under "Extended Receive and Probe" on page 3-24.

- You can use system calls to control the execution characteristics of parallel programs, as discussed under "Managing Applications" on page 4-2.

- You can open a file on all nodes at once very efficiently with **gopen()**, as discussed under "Opening Files in Parallel" on page 5-10.

- You can read the same data from a file into all nodes at the same time very efficiently with the I/O mode **M_GLOBAL**, as discussed under "Using I/O Modes" on page 5-14.

- You can read data into or write data from a series of scattered memory buffers with th **..readv()** and **...writev()** calls, as discussed under "Reading and Writing Files in Parallel" on page 5-26.

- You can find out the characteristics of PFS file systems (which are more configurable than CFS) with the **getpfsinfo()** and **statpfs()** calls, as discussed under "Getting Information About PFS File Systems" on page 5-41.

- You can use the HIPPI and FDDI network interfaces, as discussed in the *Paragon™ System High Performance Parallel Interface Manual* and *Paragon™ System Fiber Distributed Data Interface Installation and Configuration Guide.*

- You can use the Paragon application development tools to help you port and optimize your code, as discussed in the *Paragon™ System Application Tools User's Guide.*

Compilers

The Paragon system compilers work the same as the iPSC system compilers, with the following exceptions:

- The compilers, linker, and other tools are now available on the Paragon system as well as on workstations. They can be called by the standard names (**cc**, **f77**, **ld**, and so on) as well as the names used in cross-development (**icc**, **if77**, **ld860**, and so on).

- The environment variable that specifies the root of the compiler directory tree is called *PARAGON_XDEV* rather than *IPSC_XDEV*. The default for this variable is now */usr/paragon/XDEV* rather than */usr/ipsc/XDEV*.

- The compiler files are now found in the directory *$PARAGON_XDEV/paragon* rather than *$IPSC_XDEV/i860*. For example, your execution search path (*path* or *PATH* environment variable) should include the directory *$PARAGON_XDEV/paragon/bin.arch* (where *arch* identifies the architecture of the system, such as **paragon** or **sun4**) rather than *$IPSC_XDEV/i860/bin.arch* or *$IPSC_XDEV/i860/bin.*

- The **-p** switch is now ignored. See the *Paragon™ System Application Tools User's Guide* for information on profiling.

- The default for quad-alignment has been changed from **-Mnoquad** to **-Mquad**. This change results in up to four times better performance for some code.

- The new switch **-nx** has been added. This switch generates a program that automatically starts itself on multiple nodes, as discussed under "Compiling and Linking Applications" on page 2-5. The switch **-node** is currently accepted as a synonym for **-nx**, but this support may be dropped in a future release.

- You can now have a file called *.icfrc* in your home directory that defines the default compiler switches for you.

See the *Paragon™ System Fortran Compiler User's Guide* or *Paragon™ System C Compiler User's Guide* for more information on the Paragon system compilers.

NOTE

> You cannot use the Paragon system cross-compilers to produce programs for the iPSC system, and you cannot use the iPSC system cross-compilers to produce programs for the Paragon system.

If you develop programs for the iPSC system as well for the Paragon system, you must be sure that your execution search path (*PATH* or *path* variable) is set appropriately for your current target system. To compile a program for the operating system, the variable *PARAGON_XDEV* must be set appropriately and your execution search path must include *$PARAGON_XDEV/paragon/bin.arch*; to compile a program for the iPSC system, the variable *IPSC_XDEV* must be set appropriately and your execution search path must include *$IPSC_XDEV/i860/bin.arch* instead. Be sure that your execution search path does not include both these directories at the same time.

Commands

In general, all of the standard commands of UNIX System V are supported by the operating system, but none of the iPSC-system-specific commands are supported. However, many of these commands are not needed in the operating system, or have equivalent standard commands in OSF/1.

Cube Control Commands

The usage model of the Paragon system is different from that of the iPSC system. Instead of allocating a cube with a certain number of nodes, loading a program onto the cube, and then releasing the cube, you run a parallel application simply by typing its name on the command line. You can use command-line arguments to control its execution characteristics (such as the number of nodes on which it runs), and you can use standard OSF/1 process control commands such as **kill** to control the program. (See Chapter 2 for more information on running and controlling applications in the operating system.)

For this reason, the following iPSC system commands, which create and control cubes, are not supported in the operating system:

archcube This command is not needed in the operating system because all nodes currently have the same architecture.

attachcube This command is not needed in the operating system because you do not have to attach to a cube before you can use it.

cubeinfo Use the **lspart** command to list the available partitions. See "Listing Subpartitions" on page 2-60 for more information.

getcube

Use the **-sz** switch on the application command line to specify the number of nodes allocated to the application. See "Specifying Application Size" on page 2-15 for more information.

The **mkpart** command is similar to **getcube** in that it allocates a partition (a group of nodes). However, partitions are not the same as cubes: partitions can overlap, and a partition can be used by several applications at once. Depending on the policies of your site, you may or may not be allowed to allocate partitions. See "Making Partitions" on page 2-46 for more information.

killcube

Use the OSF/1 **kill** command to kill a running application, or press your interrupt key (`<Ctrl-c>` or `<Del>`). See "Managing Running Applications" on page 2-29 for more information.

load

Type an application's filename on the command line to run it on multiple nodes. See "Running Applications" on page 2-11 for more information.

newserver

This command is not needed in the operating system because you can use the usual OSF/1 I/O redirection characters to redirect an application's output. See "I/O Redirection" on page 2-12 for more information.

relcube

This command is not needed in the operating system because you do not have to release a cube once you have used it. The nodes allocated to an application are automatically released when all the processes in the application have terminated.

The **rmpart** command is similar to **relcube** in that it deallocates a partition (a group of nodes). However, partitions are not the same as cubes: partitions can overlap, and a partition can be used by several applications at once. Depending on the policies of your site, you may or may not be allowed to remove partitions. See "Removing Partitions" on page 2-52 for more information.

startcube

This command has no equivalent in the operating system. There is no way to load an application into the nodes' memory without starting it.

syslog

This command is not needed in the operating system because you can use the usual OSF/1 I/O redirection characters to redirect an application's output. The standard I/O of a node process is connected to the same files or devices as the standard I/O of its controlling process. See "I/O Redirection" on page 2-12 for more information.

waitcube

This command is not needed in the operating system because, by default, your command prompt does not return until the application has completed. Also, you can redirect the output of any program with the usual OSF/1 I/O redirection characters (see "I/O Redirection" on page 2-12 for more information).

CFS Commands

The following iPSC system commands, which control the Concurrent File System and the SRM tape drive, are not supported in the operating system:

cptape Use the **cpio** command instead. See **cpio** in the *OSF/1 Command Reference* for more information.

showvol Use the **showfs** command instead. See "Displaying File System Attributes" on page 5-5 for more information.

star Use the **tar** command instead. (Note that you must use the **-E** switch to archive a file larger than 2G–1 bytes.) See **tar** in the *Paragon™ System Commands Reference Manual* and *OSF/1 Command Reference* for more information.

stream This command is not needed in the operating system because there is no streaming tape drive.

tapemode This command currently has no equivalent in the operating system. There is no way to display or change the operating mode of the system's tape drives.

System Administration Commands

The following iPSC system commands, which are used for system administration, are not supported in the operating system:

cbackup Use the **dump** command instead. See **dump** in the *OSF/1 System and Network Administrator's Reference* for more information.

cfschk Use the **fsck** command instead. See **fsck** in the *OSF/1 System and Network Administrator's Reference* for more information.

crestore Use the **rdump** command instead. See **rdump** in the *OSF/1 System and Network Administrator's Reference* for more information.

makewhatis Use the **catman** command instead. See **catman** in the *OSF/1 Command Reference* for more information.

mkcfs Use the **newfs** command instead. See **newfs** in the *OSF/1 System and Network Administrator's Reference* for more information.

mkdev Use the **mknod** command instead. See **mknod** in the *OSF/1 System and Network Administrator's Reference* for more information.

plogon and **plogoff**
These commands currently have no equivalent in the operating system. There is currently no way to log creation and deletion of partitions or running of applications. However, you can use the **syslogd** daemon to log other system activity. See **syslogd** in the *OSF/1 System and Network Administrator's Reference* for more information.

Remote Host Commands

The following iPSC system commands, which are used for program development on remote hosts, are not supported in the operating system:

rf77 Use the **if77** command instead. See the *Paragon*™ *System Fortran Compiler User's Guide* for more information.

rcc Use the **icc** command instead. See the *Paragon*™ *System C Compiler User's Guide* for more information.

rld Use the **ld860** command instead. See the *Paragon*™ *System Fortran Compiler User's Guide* or *Paragon*™ *System C Compiler User's Guide* for more information.

ras Use the **as860** command instead. See the *Paragon*™ *System Fortran Compiler User's Guide* or *Paragon*™ *System C Compiler User's Guide* for more information.

rar Use the **ar860** command instead. See the *Paragon*™ *System Fortran Compiler User's Guide* or *Paragon*™ *System C Compiler User's Guide* for more information.

Miscellaneous Commands

The following iPSC system commands are not supported in the operating system:

less Use the **more** command instead. See **more** in the *OSF/1 Command Reference* for more information.

manpath Use the *MANPATH* environment variable instead. See **man** in the *OSF/1 Command Reference* for more information.

nsh Use the **rlogin** or **telnet** command to log into the Paragon system from your workstation. See **rlogin** or **telnet** in your workstation's documentation for more information.

rebootcube This command has no equivalent in the operating system. There is no way for ordinary users to reboot the system.

System Calls

In general, all of the standard system calls of UNIX System V and most of the iPSC-system-specific system calls are supported by the operating system. This section suggests alternatives for the unsupported calls.

NOTE

> Some iPSC calls are provided for backward compatibility only, and are not intended for use in new programs. These calls are not documented in the online manpages or in the *Paragon™ System C Calls Reference Manual* or *Paragon™ System Fortran Calls Reference Manual*. See "iPSC® and Touchstone DELTA Compatibility Calls" on page 4-66 for a list of these calls.

Include Files

The operating system does not support the iPSC system include files <*cube.h*> or <*fcube.h*>. You should replace any reference to <*cube.h*> with <*nx.h*>, and any reference to <*fcube.h*> with <*fnx.h*>.

Host Calls

Applications in the operating system do not usually have host programs. The usual programming model in the operating system is to write a single program (which corresponds to a "node program" in the iPSC system), link it with **-nx**, and execute the program on a group of nodes by typing its name (see "Running Applications" on page 2-11 for more information). You may be able to eliminate all references to the following unsupported calls by rewriting your program to use this programming model. If your application requires a separate host program, you can rewrite your host program into a *controlling process* (see "Managing Applications" on page 4-2 for more information).

For this reason, the **-host** switch to the **cc** and **f77** commands is not supported (there is no separate host library; host programs use the same library as node programs). Also, the following iPSC system calls, which are used in host programs, are not supported in the operating system:

attachcube() This call currently has no equivalent in the operating system. Unlike a host program, a controlling process cannot be associated with more than one application. Consider re-writing your host program as two or more separate programs, each of which creates one application and communicates with the other host program(s) using pipes, signals, or some other OSF/1 interprocess communication method. See "Managing Applications" on page 4-2 for information on creating and controlling applications using system calls.

cubeinfo()

This call currently has no equivalent in the operating system. However, because allocation of nodes in the operating system is not exclusive, it is not usually necessary for programs to know how other users have allocated nodes. To get information on your own application (equivalent to the "current cube"), you can use calls such as **numnodes()**.

getcube()

Use one of the **nx_initve...()** calls instead. See "Managing Applications" on page 4-2 for information on the **nx_initve...()** calls.

killcube()

This call is supported, but can only be used to kill and flush all processes on all nodes (**killcube(-1,-1)**).

You can use **kill()** to kill a single process, as discussed for **killproc()** below.

killproc()

This call is supported, but can only be used to kill all processes on all nodes (**killproc(-1,-1)**).

You can use **kill()** to kill a single process, given its OSF/1 process ID. **kill()** is supported in both C and Fortran. To determine the OSF/1 process ID of a process created by **nx_nfork()**, **nx_load()**, or **nx_loadve()**, use the values stored into the *pid_array* argument. These calls store the OSF/1 PIDs of the processes created into the elements of this array, as discussed under "Using PIDs" on page 4-20.

For example, to kill the process on node number *node*:

```
#include <signal.h>

n = nx_nfork(NULL, -1, ptype, pid_array);
    •
    •
    •
kill(pid_array[node], SIGKILL);
```

Note that process types (*ptype* in this example) in the operating system are equivalent to NX PIDs in the iPSC system. PIDs (*pid_array* in this example) in the operating system are standard UNIX process IDs.

See the *OSF/1 Programmer's Reference* for information on **kill()**; see "Managing Applications" on page 4-2 for information on **nx_nfork()**, **nx_load()**, and **nx_loadve()**.

killsyslog()

Use **freopen()** instead, to close the standard output and standard error output and reopen them to */dev/tty*. See **freopen()** in the *OSF/1 Programmer's Reference* for more information.

freopen() is not currently supported for Fortran programs. However, it is supported for C programs. You can write a C "wrapper" function, as follows:

```
#include <stdio.h>

void killsyslog_() {
    freopen("/dev/tty", "w", stdout);
    freopen("/dev/tty", "w", stderr);
}
```

Note the underscore at the end of the function name. Once you have compiled this function and linked it into your Fortran program, you can call **killsyslog()** as described in the iPSC system documentation.

newserver() This call is not necessary in the operating system. The standard I/O of a controlling process (host process) is connected to the same files or devices as the standard I/O of its node processes.

relcube() This call is not necessary in the operating system. The nodes allocated to an application are automatically released when all the processes in the application have terminated.

setpid() Use **setptype()** instead. "Process Characteristics" on page 3-3 for information on **setptype()**, and "Message Passing Between Controlling Process and Application Processes" on page 4-30 for information on using **setptype()** in a controlling process.

setsyslog() This call is not necessary in the operating system. The standard I/O of a controlling process (host process) is connected to the same files or devices as the standard I/O of its node processes.

waitall() To wait for all processes on all nodes (**waitall(-1, -1)**), call **nx_waitall()**. See "Waiting for Application Processes with nx_waitall()" on page 4-19 for more information.

To wait for a single node process (**waitall(***node***,** *pid***)**), use the OSF/1 system call **waitpid()** to wait for the process with a particular OSF/1 process ID. To determine the PID of a process created by **nx_nfork()**, **nx_load()**, or **nx_loadve()**, use the values stored into the *pid_array* argument. These calls store the OSF/1 PIDs of the processes created into the elements of this array, as discussed under "Using PIDs" on page 4-20.

For example, to wait for the process on node number *node*:

```
n = nx_nfork(NULL, -1, ptype, pid_array);
    •
    •
    •
waitpid(pid_array[node], &status, 0);
```

Note that process types (*ptype* in this example) in the operating system are equivalent to NX PIDs in the iPSC system. PIDs (*pid_array* in this example) in the operating system are standard UNIX process IDs.

See the *OSF/1 Programmer's Reference* for information on **wait()** and **waitpid()**; see "Managing Applications" on page 4-2 for information on **nx_nfork()**, **nx_load()**, and **nx_loadve()**.

wait() is supported in both C and Fortran, but **waitpid()** is not currently supported in Fortran. You can make **waitpid()** callable from Fortran by writing a C "wrapper" function, as follows:

```
#include <sys/types.h>
#include <sys/wait.h>

int waitpid_(int *process_id,
             int *status_location,
             int *options) {
    return((int)waitpid((pid_t)*process_id,
                        status_location,
                        *options);
}
```

Note the underscore at the end of the function name. Once you have compiled this file and linked it into your Fortran program, you can call **waitpid()** as described in the *OSF/1 Programmer's Reference*. The wrapper function **waitpid()** takes three **integer*4** parameters and returns an **integer*4** value.

waitone()

To wait for the first node process in the entire application to complete (**waitone(-1, -1,** *cnode***,** *cpid***,** *ccode*)), use the OSF/1 system call **wait()**. For example:

```
n = nx_nfork(nodes, NUMNODES, ptype, pids);
    •
    •
    •
pid = wait(&status);
```

After this call, the status of the first process to complete is stored in *status* and its OSF/1 process ID is stored in *pid.* To determine the process's node number, look for the value of *pid* in the *pids* array returned by **nx_nfork()**, **nx_load()**, or **nx_loadve()**.

To wait for a single node process (**waitone(***node, pid, cnode, cpid, ccode***)**), use the same technique described for **waitall(***node, pid***)**:

```
n = nx_nfork(NULL, -1, ptype, pid_array);
    •
    •
    •
pid = waitpid(pid_array[node], &status, 0);
```

In this case, the status of the process is stored in *status* and its OSF/1 process ID is stored in *pid.* To determine the process's node number, look for the value of *pid* in *pid_array* as described above.

See the *OSF/1 Programmer's Reference* for information on **wait()** and **waitpid()**; see "Managing Applications" on page 4-2 for information on **nx_nfork()**, **nx_load()**, and **nx_loadve()**. **wait()** is supported in both C and Fortran, but **waitpid()** is not; to call **waitpid()** from Fortran, use the technique discussed previously under **waitall()**.

Byte-Swapping Calls

The calls listed in Table B-1, which swap bytes between the format used on the cube and the format used on some remote hosts, are not supported in the current release of the operating system.

Table B-1. Unsupported iPSC® System Byte-Swapping Calls

createstruc()	**CTOHF()**	**HTOCC()**	**HTOCL()**
CTOHC()	**CTOHL()**	**HTOCD()**	**HTOCS()**
CTOHD()	**CTOHS()**	**HTOCF()**	**relstruc()**

You can use the standard OSF/1 system calls **htonl()**, **htons()**, **ntohl()**, and **ntohs()** to swap bytes between the standard format for your machine and the Internet network format. See **htonl()**, **htons()**, **ntohl()**, and **ntohs()** in the *OSF/1 Programmer's Reference* for more information.

htonl(), **htons()**, **ntohl()**, and **ntohs()** are not currently supported for Fortran programs. However, they are supported for C programs. You can make them callable from Fortran by writing C "wrapper" functions, as follows:

```
#include <netinet/in.h>

long htonl_(long *hostlong) {
    return((long)htonl((unsigned long)*hostlong);
}

short htons_(short *hostshort) {
    return((short)htons((unsigned short)*hostshort);
}

long ntohl_(long *netlong) {
    return((long)ntohl((unsigned long)*netlong);
}

short ntohs_(short *netshort) {
    return((short)ntohs((unsigned short)*netshort);
}
```

Note the underscore at the end of each function name. Once you have compiled this file and linked it into your Fortran program, you can call **htonl()**, **htons()**, **ntohl()**, and **ntohs()** as described in the *OSF/1 Programmer's Reference*. The wrapper functions **htonl()** and **ntohl()** take an **integer*4** parameter and return an **integer*4** value; the wrapper functions **htons()** and **ntohs()** take an **integer*2** parameter and return an **integer*2** value.

Floating-Point Control Calls

The C system calls **fpgetsticky()** and **fpsetsticky()**, which get and set the i860 microprocessor's *floating-point exception sticky flags*, and **fpgetmask()** and **fpsetmask()**, which get and set the *floating-point exception mask*, do not support the exception value **FP_X_DNML**, which represents a denormalization exception in the iPSC system.

The Fortran system call **fpsetmask()** also does not support the denormalization exception, and uses different numeric values to represent the various exceptions than the corresponding iPSC system call. See "Controlling Floating-Point Behavior" on page 4-60 for the correct values for the operating system.

CFS Calls

In the operating system, the iPSC system's Concurrent File System (CFS) has been replaced by the Parallel File System (PFS). PFS calls are compatible with CFS calls; however, PFS offers additional functionality (see Chapter 5 for more information). This section lists the differences that may affect some programs that use CFS calls.

iread() and **iwrite()**

These calls work in the operating system just as they do in the iPSC system. In both systems, the number of I/O IDs is limited; however, the limit in the operating system is much smaller than in the iPSC system. (In the iPSC system the limit is 5000; in the operating system it is at least 256, but may vary from release to release.) For this reason, it is very important that you use **iodone()** or **iowait()** to release each ID as soon as possible after you use it. If you program in C, you can use **_iread()** or **_iwrite()** to detect the "too many requests" error (**EQNOMID**).

open()

Many iPSC system programs use code like the following to open a file on all nodes:

```
if(mynode() == 0) {
    fd = open("myfile", O_CREAT | O_RDWR, 0644);
    gsync();
} else {
    gsync();
    fd = open("myfile", O_RDWR, 0644);
}
setiomode(fd, iomode);
```

The **open()** call works the same in the operating system as it does in the iPSC system. However, if this code is executed on many nodes, the large number of **open()** requests arriving simultaneously at the I/O node can cause the I/O node to slow down, hang, or even crash. This can even cause the system to crash.

You should always use the **gopen()** call instead of this type of code. For example, you should replace the lines shown above with the following:

```
fd = gopen("myfile", O_CREAT | O_RDWR,
           iomode, 0644);
```

gopen() opens a file simultaneously on all nodes and sets its I/O mode in a single operation. It is much more efficient than having each node call **open()**, and avoids this type of system crash completely.

Note that **gopen()** opens the same file on each node. If each node is opening its own file, you must still use **open()**. However, you should try to avoid using **open()** together with **gsync()**, to prevent all the **open()** requests from arriving at the I/O node at the same time.

Miscellaneous Calls

The following iPSC system calls work differently or are not supported in the operating system:

dclock() This call works in the operating system just as it does in the iPSC system: it returns the time since the system was booted, in seconds. However, in a gang-scheduled partition your application may be *rolled out* and then *rolled in* again. While it is rolled out, the application is stopped but the **dclock()** clock keeps going (reflecting "wall-clock" time), which means that **dclock()** cannot be used to determine the amount of time your application has actually been running.

Using **dclock()** in a gang-scheduled partition may result in incorrect MFLOPS estimates. You can use the **time** command, **getrusage()** system call (C only), or the **etime()** or **dtime()** routine (Fortran only) instead to determine your application's CPU usage. See the *OSF/1 Command Reference* for information on **time**, the *OSF/1 Programmer's Reference* for information on **getrusage()**, and the *Paragon™ System Fortran Compiler User's Guide* for information on **etime()** and **dtime()**.

flushmsg() This call currently has no equivalent in the operating system. It may be supported in a future release.

getiphosts() This call currently has no equivalent in the operating system. However, because the OSF/1 operating system automatically routes network traffic using all available Ethernet ports, it is not usually necessary to know the network names of the available ports.

gixor() This call is not supported in the operating system. The exclusive OR operator is not associative, and gives unpredictable results when used on more than two nodes.

glxor() This call is not supported in the operating system. The exclusive OR operator is not associative, and gives unpredictable results when used on more than two nodes.

handler() Use the **signal()** system call instead (**signal()** is supported for both C and Fortran). See **signal()** in the *OSF/1 Programmer's Reference* for information on signal handling; see **signal()** in the *Paragon™ System Fortran Compiler User's Guide* for information on the Fortran interface to **signal()**.

plogon() and **plogoff()**

These calls currently have no equivalent in the operating system. There is currently no way to automatically log creation and deletion of partitions or running of applications. However, you can use the **syslog()** call to log activities under program control. See **syslog()** in the *OSF/1 Programmer's Reference* for more information.

setiphost() This call is not necessary in the operating system. The OSF/1 operating system automatically routes network traffic using all available Ethernet ports; it is not necessary to select one port to perform network operations.

setpgrp() There are two different versions of this call in the operating system. The standard version of **setpgrp**(), found in *libbsd.a*, is equivalent to **setpgid**() and is not compatible with the iPSC/860 version. The System V version of **setpgrp**(), found in *libsys5.a*, is equivalent to **setsid**() and is compatible with the iPSC/860 version. To get the iPSC/860-compatible version, be sure to use the switch **-lsys5** when linking.

Summary

Table B-2 summarizes the operating system equivalents for the unsupported iPSC system commands.

Table B-2. Summary of Unsupported iPSC® System Commands (1 of 2)

iPSC® System Command	Operating System Equivalent
archcube	(none)
attachcube	(none)
cbackup	**dump**
cfschk	**fsck**
cptape	**cpio**
crestore	**rdump**
cubeinfo	**lspart**
getcube	**-sz** switch on application command line
killcube	**kill**
less	**more**
load	Application's filename
makewhatis	**catman**
manpath	*MANPATH* environment variable
mkcfs	**newfs**
mkdev	**mknod**
newserver	I/O redirection characters
nsh	**rlogin** or **telnet**

Table B-2. Summary of Unsupported iPSC® System Commands (2 of 2)

iPSC® System Command	Operating System Equivalent
plogoff	(none)
plogon	(none)
rar	**ar860**
ras	**as860**
rcc	**icc**
rebootcube	(none)
relcube	(none)
rf77	**if77**
rld	**ld860**
showvol	**showfs**
star	**tar**
startcube	(none)
stream	(none)
syslog	I/O redirection characters
tapemode	(none)
waitcube	(none)

Table B-3 summarizes the operating system equivalents for the unsupported iPSC system calls.

Table B-3. Summary of Unsupported iPSC® System Calls (1 of 2)

iPSC® System Call	Operating System Equivalent
attachcube()	(none)
cubeinfo()	(none)
dclock()	Supported, but use **getrusage()** (C) or **etime()/dtime()** (Fortran) to determine CPU usage in gang-scheduled partitions.
fpgetsticky(), **fpsetsticky()**, **fpgetmask()**, **fpsetmask()**	Supported, except for **FP_X_DNML**, but Fortran mask values are different.
flushmsg()	(none)
getcube()	**nx_initve...()**

Table B-3. Summary of Unsupported iPSC® System Calls (2 of 2)

iPSC® System Call	Operating System Equivalent
getiphosts()	(none)
gixor()	(none)
glxor()	(none)
handler()	**signal()**
iread(), **iwrite()**	Supported, but number of I/O IDs is much smaller.
killcube()	Use **killcube(-1,-1)** to kill and flush all processes; use **kill()** to kill one process
killsyslog()	(none)
killproc()	Use **killproc(-1,-1)** to kill all processes; use **kill()** to kill one process
newserver()	**freopen()**
open()	Supported, but use **gopen()** instead if possible.
plogoff()	(none)
plogon()	(none)
relcube()	(none)
setiphost()	(none)
setpgrp()	Supported, but be sure to link with **-lsys5** to get the correct version.
setpid()	**setptype()**
setsyslog()	(none)
waitall()	Use **nx_waitall()** to wait for all processes; use **wait()** or **waitpid()** to wait for one process
waitone()	**wait()** or **waitpid()**
Byte-swapping calls	**htonl()**, **htons()**, **ntohl()**, and **ntohs()**

Index

intel

Paragon™ System User's Guide