April 1993

Order Number: 312489-001

PARAGON™ OSF/1 USER'S GUIDE

Intel® Corporation

REV.	REVISION HISTORY	DATE
-001	Original Issue	4/93

LIMITED RIGHTS

Preface

This manual tells how to use the Paragon™ OSF/1 operating system on an Intel 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.

Organization

Chapter 1	Provides an overview of the Paragon OSF/1 software and Intel supercomputer hardware.
Chapter 2	Describes the Paragon OSF/1 commands that you can enter at the shell prompt and the Paragon OSF/1 cross-development commands that run on supported workstations.
Chapter 3	Describes the message-passing system calls available to programs in Paragon OSF/1.
Chapter 4	Describes the other general-purpose system calls available in Paragon OSF/1.
Chapter 5	Describes the parallel I/O calls you can use for parallel access to the Intel supercomputer's file systems.
Chapter 6	Tells how to prepare an application for the Paragon OSF/1 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.

Appendix A	Summarizes the commands and system calls of Paragon OSF/1. 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 Paragon OSF/1 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:

Paragon™ Manuals

- *Paragon™ OSF/1 Commands Reference Manual*
- *Paragon™ OSF/1 C Compiler User's Guide*
- *Paragon™ OSF/1 Fortran Compiler User's Guide*
- *Paragon™ OSF/1 C System Calls Reference Manual*
- *Paragon™ OSF/1 Fortran System Calls Reference Manual*
- *Paragon™ OSF/1 Software Tools User's Guide*
- *Paragon™ OSF/1 Interactive Parallel Debugger Manual*
- *Paragon™ XP/S i860™ 64-Bit Microprocessor Assembler Reference Manual*

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 Supercomputer 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

Intel Corporation Italia s.p.a.
Milanofiori Palazzo
20090 Assago
Milano
Italy
1678 77203 (toll free)

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

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

United Kingdom Intel Corporation (UK) Ltd.
Supercomputer System Division
Pipers Way
Swindon SN3 IRJ
England
0800 212665 (toll free)
(44) 793 491056 (*answered in French*)
(44) 793 431062 (*answered in Italian*)
(44) 793 480874 (*answered in German*)
(44) 793 495108 (*answered in English*)

Germany Intel Semiconductor GmbH
Dornacher Strasse 1
8016 Feldkirchen bel Muenchen
Germany
0130 813741 (toll free)

World Headquarters
Intel Corporation
Supercomputer Systems Division
15201 N.W. Greenbrier Parkway
Beaverton, Oregon 97006
U.S.A.
(503) 629-7600

If you have comments about the Paragon 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 2 Using Paragon™ OSF/1 Commands

Chapter 3 Using Paragon™ OSF/1 Message-Passing System Calls

Chapter 5 Using Parallel File I/O

Chapter 6 Designing a Parallel Application

Appendix A Summary of Commands and System Calls

List of Illustrations

List of Tables

List of Tables

Introduction 1

Introduction

This chapter introduces the Paragon™ OSF/1 operating system and the hardware it runs on.

In an Intel supercomputer, a large number of processors called *nodes* work concurrently on the parts of a problem. Each node can run multiple processes, and each process can have multiple *threads* (lightweight processes). The processes and threads on each node time-share the node's processor, 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 Paragon OSF/1 system calls.

The processes in an application can also share disk files; Paragon OSF/1 *parallel I/O calls* insure that access to these files is efficient and properly synchronized.

System Hardware

The Paragon OSF/1 operating system runs on several models of Intel 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 one or more 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. An Intel supercomputer can have up to 2000 nodes. Each node can run more than one process at the same time; these processes can belong to the same or different applications.

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*. However, there are no physical differences between these two types of 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.

System Software

The nodes run the *Paragon OSF/1* operating system, 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; Paragon OSF/1 is an extended version of OSF/1 with enhancements to support parallel processing.

The Intel supercomputer also comes with a *cross-development facility*, which you can use to compile and link Paragon OSF/1 programs on supported workstations.

Paragon™ OSF/1 Operating System

Paragon OSF/1 provides all the standard features of OSF/1, 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 Paragon OSF/1 operating system provides message passing capability, Parallel File System™ access, and various other utilities to programs running on the Intel supercomputer. With Paragon OSF/1 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 Intel supercomputer's 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 Paragon OSF/1 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 Intel supercomputer.

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

You compile and link your application with the self-hosted Paragon OSF/1 compilers and linker. You then execute your application on the nodes of the Intel 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 Paragon OSF/1 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™ OSF/1 Software Tools User's Guide.*

Programming Model

The most common programming model used with Paragon OSF/1 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

Paragon OSF/1 comes with a complete program development environment, including compilers, linker, libraries, and related tools. You can perform all phases of program development on the Intel supercomputer. In addition, the compilers, linker, and libraries for Paragon OSF/1 are also available on selected UNIX workstations. This *cross-development facility* lets you edit, compile, and link Paragon OSF/1 programs on your own workstation.

Note, though, that the cross-development facility does not include a way to run a Paragon OSF/1 executable that resides on your workstation's disk. You must transfer your executable files to the Intel supercomputer for execution and debugging. You can do this by mounting your workstation's file system onto the Intel supercomputer, or the Intel 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 Intel supercomputer. To execute files on the Intel supercomputer once they are transferred, you can use the standard **rsh** or **rcmd** command.

Using Paragon™ OSF/1 Commands 2

Introduction

This chapter tells you how to use Paragon OSF/1 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 *System Administrator's Guide* for your system for information on commands that require root privilege.

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 Intel 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, but this allocation is not exclusive (other applications can run on the same nodes).

All Intel 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 site's policies.

Using Paragon™ OSF/1 Commands on the Intel® Supercomputer

The Paragon OSF/1 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.

Paragon OSF/1 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™ OSF/1 Commands Reference Manual.*

To use any of these commands, you must first log into an Intel supercomputer. Intel 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 Intel 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 Paragon™ OSF/1 Commands on Workstations

The Paragon OSF/1 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™ OSF/1 C Compiler User's Guide* and *Paragon™ OSF/1 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 Paragon OSF/1 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 an Intel 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 Intel 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 an Intel supercomputer.

Information You Need

Before you begin, you will need the following information:

- The network name of your Intel supercomputer.
- The command to use to log into the Intel supercomputer, such as **rlogin** or **telnet**.
- Your user name and password on the Intel 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 Intel 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, Paragon OSF/1 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 a Paragon OSF/1 application written in C on an Intel supercomputer.
f77 -nx [*switches*] *sourcefile...*	Compile a Paragon OSF/1 application written in Fortran on an Intel supercomputer.
icc -nx [*switches*] *sourcefile...*	Compile a Paragon OSF/1 application written in C on an Intel supercomputer or cross-development workstation.
if77 -nx [*switches*] *sourcefile...*	Compile a Paragon OSF/1 application written in Fortran on an Intel supercomputer or cross-development workstation.

You can compile and link applications on the Intel supercomputer itself, or on a workstation that supports the Paragon OSF/1 cross-development environment. On the Intel 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 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.)
- 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.

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

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

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

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

NOTE

Do not use **-nx** if your application calls **nx_initve()**.

The Paragon OSF/1 operating system provides **nx_initve()** 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 calls **nx_initve()** itself, the application's call to **nx_initve()** will fail and return -1. See "Controlling Application Execution" 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 Paragon OSF/1 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 Paragon OSF/1 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 Paragon OSF/1 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 Intel supercomputer. You can do this by putting it in a directory that is shared between your workstation and the Intel supercomputer with the Network File System (NFS), or by copying it to the Intel 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 command ***chmod +x myapp*** on the Intel supercomputer to give *myapp* execute permission.

NOTE

The Paragon OSF/1 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 for Paragon OSF/1, 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 Paragon OSF/1, 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 Paragon OSF/1 applications (on either the Intel 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™ OSF/1 C Compiler User's Guide.*
f77, **if77**	*Paragon™ OSF/1 Fortran Compiler User's Guide.*

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

As a general rule, always include the file *nx.h* in all Paragon OSF/1 C programs. This file contains definitions and declarations needed by the Paragon OSF/1 C system calls. Although a specific application may not need the definitions and declarations contained in *nx.h*, the overhead involved in including it in all programs is minor. Include it in your C programs as follows:

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

For Fortran programs, the corresponding file is *fnx.h*. Include it 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 Intel 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 in 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 at a site where the Paragon OSF/1 libraries are located in the directory */usr/local/lib*:

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

- Specify the complete pathname of the appropriate library or libraries on the command line. 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*:

  ```
  % 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 **-L** and **-l** switches and for listing libraries when linking. When constructing command lines, keep the following guidelines in mind:

- List libraries in the order in which they should be searched. The Paragon OSF/1 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). Backward references between objects, 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 a Paragon OSF/1 executable file (and, if necessary, copied the executable to an Intel supercomputer), you run it by typing its name at your Paragon OSF/1 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
```

If you get unexpected error messages such as "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 2-14 for more information on these variables; see your system administrator for information on the proper settings for these variables at your site.

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 all the nodes of 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 "Controlling Application Execution" 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 all the nodes of 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.

I/O Redirection

You can redirect the standard input, standard output, and standard error output 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 output 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 5-6 for more information. The standard input, standard output, and standard error output 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.

Controlling the Application's Execution Characteristics

Command Synopsis	Description
application [**-sz** *size*] [**-pri** *priority*] [**-pt** *ptype*] [**-on** *nodespec*] [**-pn** *partition*] [*mp_switches*] [\; *app2* [**-pt** *ptype*] [**-on** *nodespec*]] ...	Execute a Paragon OSF/1 application.

When you run an application, you can use command-line switches and environment variables to control the way the application executes. The 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 "Controlling Application Execution" 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 Intel 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 Intel 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 2-20) to run an application in a different partition.

For more information about partitions, see "Managing Partitions" on page 2-24.

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 Intel 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 Intel 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 2-27 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. However, this allocation is not 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
```

If you don't use the **-sz** switch, the application's size is specified by the environment variable *NX_DFLT_SIZE*. 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. The value of *NX_DFLT_SIZE* must be a positive integer less than or equal to the number of nodes in the partition. If *NX_DFLT_SIZE* is not set, the application runs on all nodes of the partition, and its size is set to the size of the partition.

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*, **numnodes**() returns 64 and **mynode**() returns a number from 0 to 63 inclusive. There is no way for an application to change its size.

The nodes allocated to the application will not necessarily be *contiguous* (that is, they may not all be physically next to each other). However, the node numbers within the application, as returned by **mynode**(), will always be sequential from 0.

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 2-32 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 4-7 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. A process can change its process type by calling **setptype()**. However, once a process has used a process type, 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 2-18. 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**. 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 "Controlling Application Execution" 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, allocate a partition containing those nodes and run the application on all nodes of that partition (see "Managing Partitions" on page 2-24 for information on partitions).

If you use **-on** when you should be using **-sz**, 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 operations will hang. (The global operations described under "Global Operations" on page 3-29, such as **gdsum()**, 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.)

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**, **-pri**, **-pt**, **-on**, **-pn**, and *mp_switches*) with the first file. The effect of these switches varies according to the switch:
    - The **-sz**, **-pri**, **-pn**, and *mp_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.
- You can use only the **-pt** and **-on** switches with the second and subsequent files. 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
```

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 in the specified partition, you may get a "partition resources exceeded" 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 2-27 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 2-24.

Specifying Message-Passing Configuration Parameters

You can control the values of some important message-passing configuration parameters for your application with the following switches, which are known as the *mp_switches*. Although the default values of these switches have been chosen to give good results for most applications, you may be able to improve your application's message-passing performance by using different values.

In general, you should begin by changing only the *memory_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 *memory_buffer*, the defaults for the other parameters are automatically scaled to match the current *memory_buffer* size. Increasing the *memory_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 *memory_buffer* size for your application, you can change the other parameters to fine-tune the usage of memory within the *memory_buffer* and optimize message-passing performance.

The values used with the *mp_switches* (except **-plk**) are integer numbers of bytes. If the value you specify is not a multiple of 32, it is silently rounded down to a multiple of 32.

-pkt *packet_size*	Sets the size of each packet. If a message is larger than *packet_size*, it is sent in several pieces, each *packet_size* bytes long.
-mbf *memory_buffer*	Sets the total amount of memory allocated to message buffers in each process.
-mex *memory_export*	Sets the total amount of memory allocated to buffering messages from other nodes. Memory in *memory_buffer* outside of *memory_export* is used for local messages (those sent and received within the same process).
-mea *memory_each*	Sets the amount of memory allocated to buffering messages from each other node. Memory in *memory_export* outside of **numnodes**() times *memory_each* is used for buffering messages from any sender, when needed.
-sth *send_threshold*	Sets the threshold for sending multiple packets. If a sender knows that it has at least *send_threshold* bytes of memory free in its *memory_each* segment on the receiving node, it will send multiple packets of a message right away. Otherwise, it will send one packet and wait for an acknowledgment that a receive has been posted.
-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. A receiving node tells its senders how much free memory the sender has in its *memory_each* segment 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. If the receiver knows that the 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.

-plk

Locks the entire data area of each process into memory, like the OSF/1 system call **plock**(). See the *OSF/1 Programmer's Reference* for information on **plock**(). **-plk** also conditions message-passing code to run more efficiently by assuming that all data buffers are locked into memory.

The default, maximum, and minimum values for these switches are shown in Table 2-1.

Table 2-1. Message-Passing Configuration Switches

Switch	Parameter	Default	Maximum	Minimum
-pkt	*packet_size*	1792 or ((*memory_each* / 2) - **sizeof(xmxg_t)***), whichever is less	1792	32
-mbf	*memory_buffer*	1MB + (10 * *full_packet_size*†) for local messages	available physical memory	2 * (2 * *full_packet_size* * **numnodes**() + 2) + (10 * *full_packet_size*) for local messages
-mex	*memory_export*	*memory_buffer* - (10 * *full_packet_size*)	*memory_buffer* - (10 * *full_packet_size*)	2 * (**numnodes**() + 2) * minimum *memory_each*
-mea	*memory_each*	(10 * *full_packet_size*) or maximum *memory_each*, whichever is less	(*memory_export* / 2) / (**numnodes**() + 2)	2 * *full_packet_size*
-sth ‡	*send_threshold*	*memory_each* / 2	*memory_each* - 1	(no minimum)
-sct ‡	*send_count*	*memory_each* / 2	*memory_each*	*packet_size*
-gth ‡	*give_threshold*	*packet_size*	*memory_each* / 2	*packet_size*

* **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 32 bytes.

† *full_packet_size* = *packet_size* + **sizeof(xmxg_t)**.

‡ All values for this parameter are silently rounded down to the nearest multiple of *packet_size*.

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 are often *gang-scheduled* for better performance and 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. The rollin quantum can be anything from a fraction of a second to 24 hours. 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 2-32 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 "Controlling Application Execution" 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**. Of these two processes, the one with the higher PID is the controlling process. The process with the lower PID is another special process, called the *shepherd process*. This process is necessary for the application; do not kill it. When the application terminates, this process will terminate as well.

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 2-47 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™ OSF/1 Interactive Parallel Debugger Manual* for information on **ipd**.

Managing Partitions

The nodes of the Intel 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 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 Intel 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 Intel 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.

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.

The Root Partition

The *root partition* is the basis for all other partitions. The name of the root partition is . (dot). The root partition always uses gang scheduling (see "Gang Scheduling" on page 2-34 for more information).

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 is always rectangular in shape. It 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.

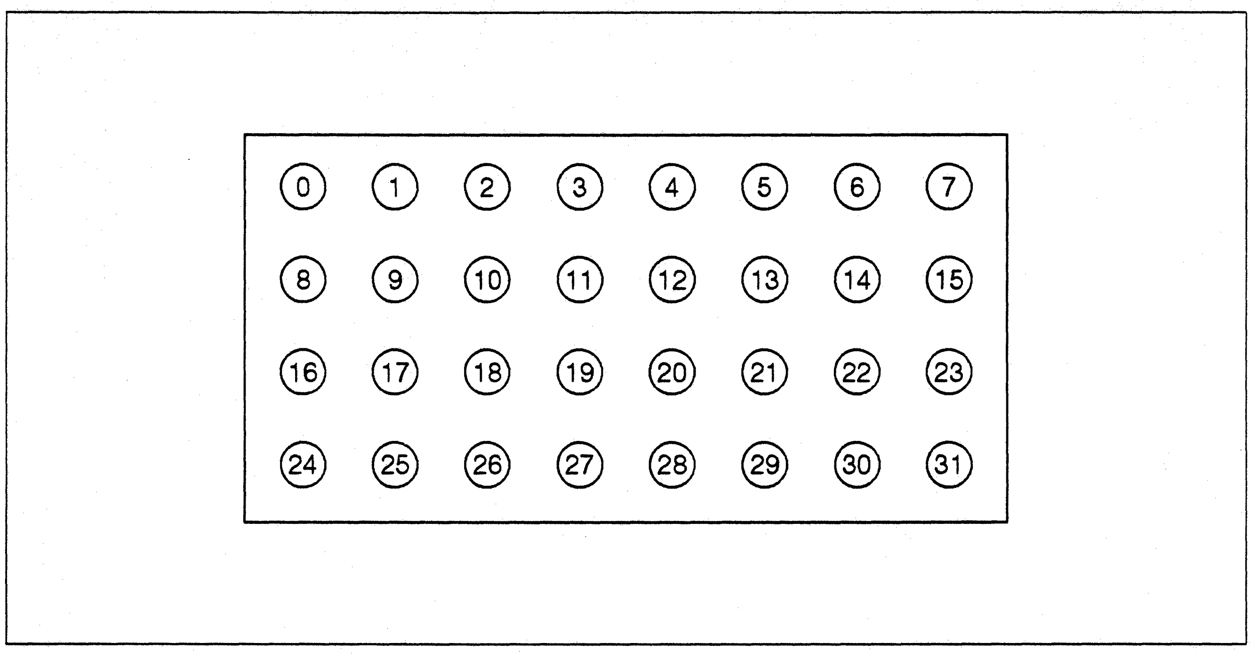


Figure 2-1. The Root Partition of a 32-Node System

NOTE

The root partition always has a mesh structure, even if the underlying hardware does not.

The Paragon OSF/1 operating system presents the same view of the system even if the node interconnect network has a different architecture (such as a hypercube).

NOTE

The root partition is always rectangular, even if the number of usable nodes in it is not a multiple of two numbers. (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 2-31. 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 always uses standard scheduling, which means that it may not contain any subpartitions (see "Standard Scheduling" on page 2-33 for more information).

When you log into the Intel 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. Running processes may also be migrated to other nodes to improve load balancing.

The Compute Partition

The *compute partition* is the partition in which parallel applications run. The name of the compute partition is *compute*. The compute partition always uses gang scheduling.

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 2-38, 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 2-44, 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 2-48.

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. This allocation is not exclusive: some or all of these nodes may also be allocated to other partitions. 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.

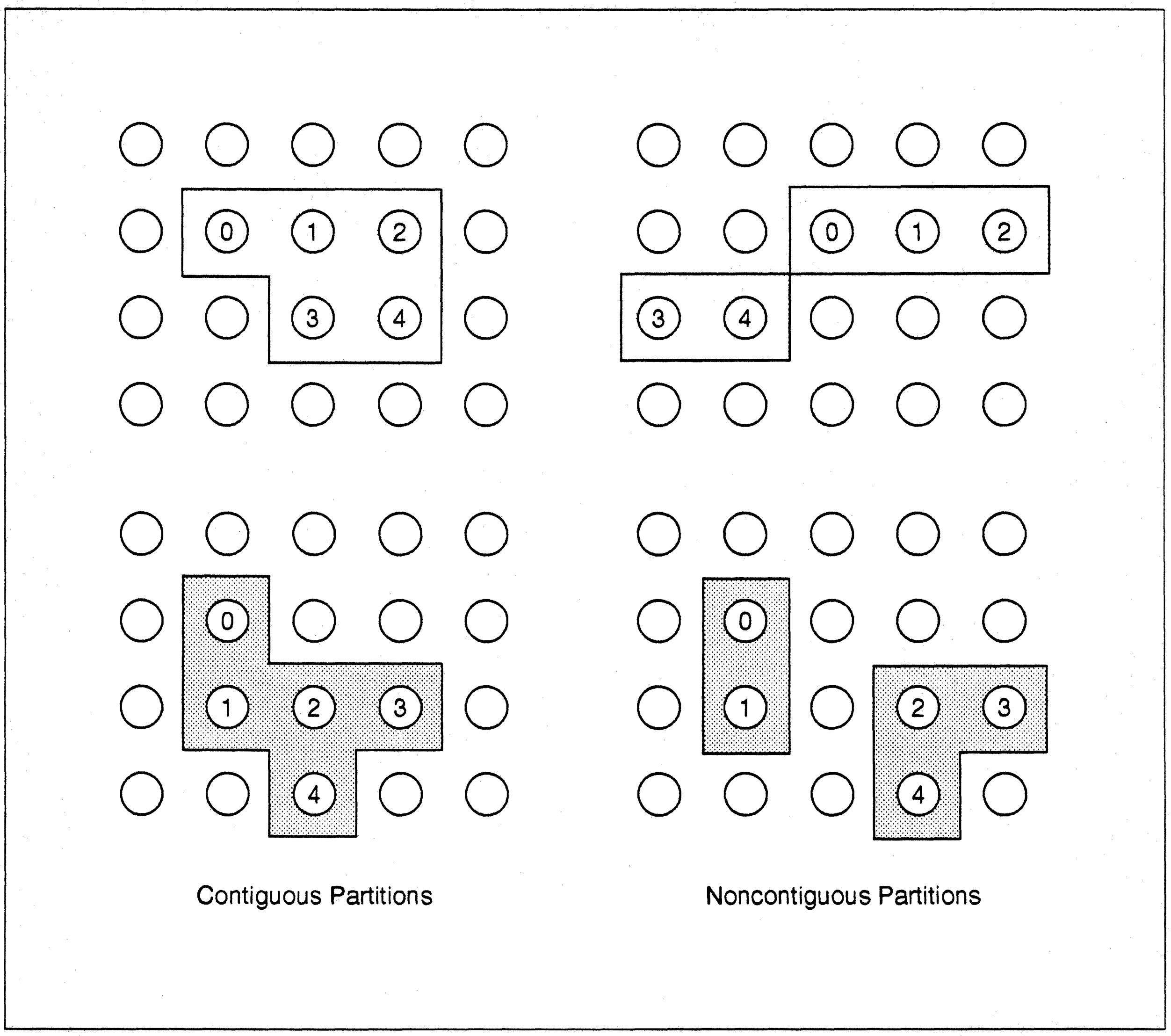


Figure 2-2. Node Numbers in Contiguous and Noncontiguous Partitions

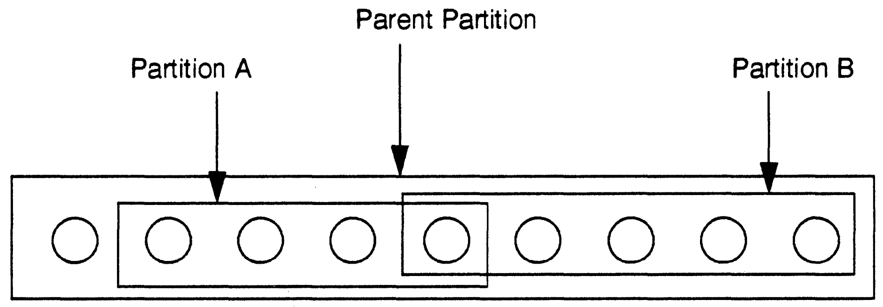
Node Numbers Within a Partition

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. (Exception: if you run an application on "all nodes" of this partition, which occurs if you don't use the **-sz** switch and the environment variable *NX_DFLT_SIZE* is not defined, the application will run on 19 nodes. This is *not* currently true if you attempt to make a subpartition containing "all nodes" of the partition, which is the default.)

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 two different forms of scheduling: *standard scheduling* and *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 *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; gang-scheduled partitions should be used to run numerically-intensive applications that do not interact with the user.

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

Standard Scheduling

Standard scheduling is the same as the scheduling technique used on single-processor OSF/1 systems. Each node in a partition that uses standard scheduling 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. Non-parallel processes in standard-scheduled partitions may be *migrated* by the system (moved from one node to another within the partition while they run) to improve load balancing; processes that are part of a parallel application do not migrate.

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*.

Gang Scheduling

Gang scheduling is a special scheduling technique that coordinates the scheduling of related processes running on separate processors. 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. Processes in partitions that use gang scheduling do not migrate.

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.

For example, Figure 2-4 shows a partition (called Partition X) that contains two applications (called Applications A and B) and one partition (called Partition Y). Partition Y is a subpartition of Partition X, and Partition X is the parent partition of Partition Y. Partition Y contains two applications (Applications C and D). Application A does not overlap any other entity; Application B and Partition Y overlap. Applications C and D overlap each other within Partition Y. The priorities and effective priority limits of these entities are shown in the figure; the rollin quanta of the two partitions are the same.

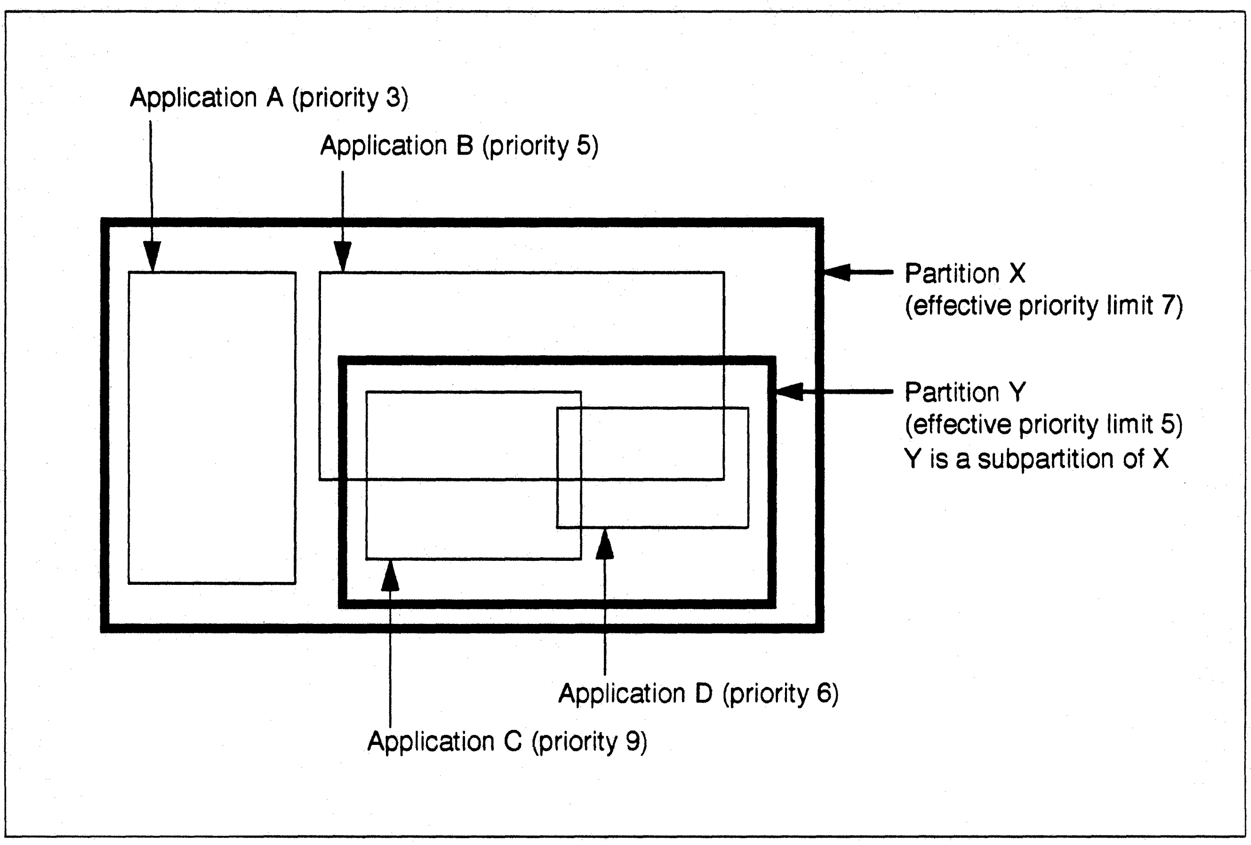

Application A (priority 3)
Application B (priority 5)
Partition X
(effective priority limit 7)
Partition Y
(effective priority limit 5)
Y is a subpartition of X
Application D (priority 6)
Application C (priority 9)

Figure 2-4. An Example of Gang Scheduling

1. At the beginning of the first rollin quantum, the system notices that Application A does not overlap with anything else, so it is simply scheduled to run for this rollin quantum regardless of its priority. However, Application B and Partition Y overlap, so the system compares their priorities:

   - Application B has a priority of 5.

   - The highest priority within Partition Y is 9. However, the partition's effective priority limit is only 5, so Partition Y has a priority of 5.

   Since Application B and Partition Y have the same priority, they will be scheduled to run in alternation. The system arbitrarily selects Application B to run first, so in the first rollin quantum Applications A and B are active and Partition Y (with Applications C and D) is inactive.

2. At the beginning of the second rollin quantum, Application A still does not overlap anything, so it is scheduled to run again in this rollin quantum. However, Application B has had its turn, so Partition Y is scheduled to run in this rollin quantum instead. This means that the entities within Partition Y must be scheduled. Applications C and D overlap, so their priorities are

compared. Since Application C's priority is higher, it is selected to run in this rollin quantum. So in the second rollin quantum Applications A and C are active and Applications B and D are inactive.

3. The third rollin quantum (and every odd-numbered rollin quantum thereafter) is scheduled like the first; Applications A and B are active.

4. The fourth rollin quantum (and every even-numbered rollin quantum thereafter) is scheduled like the second; Applications A and C are active.

Since Application A does not overlap any other entities, it is allowed to run in every rollin quantum until it completes. Applications B and C run in alternate rollin quanta until one of them completes. Since Application D has a lower priority than Application C, it does not get any processor time at all until Application C completes. If Application B is still running when Application C completes, Application D alternates with Application B just as Application C did.

NOTE

If the rollin quantum of a subpartition is larger than that of its parent partition, it will take more than one actual rollin period to satisfy the subpartition's "virtual rollin quantum."

For example, suppose the rollin quantum of Partition X in Figure 2-4 is 5 seconds and the rollin quantum of Partition Y is 10 seconds.

1. At time 0:00, the system examines Partition X and selects Application B to run in alternation with Partition Y. Application B is rolled in first and runs for the 5-second rollin quantum specified by its partition (Partition X).

2. At time 0:05, Application B is rolled out and Partition Y is rolled in for the first time. The system now examines Partition Y and selects Application C to run. Application C runs for the first 5 seconds of the 10-second rollin quantum specified by its partition (Partition Y).

3. At time 0:10, Partition Y is rolled out; this pauses Application C in the middle of its rollin quantum. Application B is rolled in, and runs for another 5-second rollin quantum.

4. At time 0:15, Application B is rolled out and Partition Y is rolled in again. Application C now runs for the last 5 seconds of its 10-second rollin quantum. Note that Partition Y does not undergo scheduling at this time. Instead, it simply continues what it was doing when it was rolled out at time 0:10.

As discussed earlier, during this period Application A runs in every rollin quantum and Application D does not run at all.

Making Partitions

Command Synopsis	Description
mkpart [**-sz** *size* | **-sz** *hXw* | **-nd** *nodespec*] [**-ss** | [[**-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 three ways to specify which nodes are allocated to the new partition:

-sz *size*	Creates a partition whose size (number of nodes) is *size*. The nodes are not necessarily contiguous.
-sz *hXw*	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*. 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.

If you don't use the **-sz** or **-nd** switch, all the nodes of the parent partition are allocated to the new partition (if the parent partition is the root partition and it contains unusable nodes, only the usable nodes are allocated). You can use at most one **-sz** or **-nd** switch in a single **mkpart** command.

The following examples all create a 50-node 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).

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 2-29. 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.

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 **-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.

No matter how you specify it, the rollin quantum must not be more than 24 hours.

-rq can be used with or without **-epl**. If you use **-rq** without **-epl**, the new partition has the same effective priority limit as its parent partition.

-epl *priority*	Creates a partition that uses gang scheduling 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 can be used with or without **-rq**. If you use **-epl** without **-rq**, the new partition has the same rollin quantum as its parent partition.

If you don't use the **-ss**, **-rq**, or **-epl** 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 gang scheduling with an effective priority limit of 7 and the same rollin quantum as its parent partition:

```
% mkpart -epl 7 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 [*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                  RQ    EPL
smith     eng          777     9                  15m     5
     +---------+
    0| . . . . |
    4| . * * * |
    8| . * * * |
   12| . * * * |
     +---------+
```

In this case, *mypart* belongs to user *smith* in group *eng*. It has permissions 777 (rwxrwxrwx), a size of 9 nodes, a rollin quantum of 15 minutes, and an effective priority limit of 5. See "Partition Characteristics" on page 2-28 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 2-31 for more information.

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

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

Listing Subpartitions

Command Synopsis	Description
lspart [**-r**] [*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              RQ   EPL  PARTITION
 chris    eng          777    16             15m     3  mandelbrot
 chris    eng          777    16               -     -  debug
 pat      mrkt         755     4             10m    10  slalom
 smith    eng          700     *               *     *  private
```

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, a size of 16 nodes, 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. It has no rollin quantum or effective priority limit: this shows that it is a standard-scheduled partition.
- *slalom* is owned by user *pat* in group *mrkt*; it has permissions rwxr-xr-x, a size of 4 nodes, a rollin quantum of 10 minutes, and an effective priority limit of 10.
- *private* is owned by user *smith* in group *eng*; it has permissions rwx------. Because *private* does not grant you read permission, its size, rollin quantum, and effective priority limit 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         RQ          EPL  PARTITION
 chris    eng          777    14 / 16   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 2-31 for more information.

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

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

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            RQ    EPL  PARTITION
.compute.mypart:
   chris     eng          777    16            15m     3  mandelbrot
   chris     eng          777    16             -      -  debug
   pat       mrkt         755     4            10m    10  slalom
   smith     eng          700     *             *      *  private
.compute.mypart.mandelbrot:
   chris     eng          777    16            15m    10  hi_pri
   chris     eng          777    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 the Applications in a Partition

Command Synopsis	Description
pspart [*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    TIME ACTIVE   TOTAL TIME COMMAND
    12345 pat          256   5     45.00  75%      4:41.60 /home/pat/glide
    23456 chris         67   4         -    -      7:12.30 boggle -sz 67
    34567 smith        192  10   1:00.00 100%   2:12:03.90 myfft -sz 192
```

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 4-14 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 2-14 for more information).
PRI	The application's priority (see "Specifying Application Priority" on page 2-15 for more information).

`TIME ACTIVE`	The amount of time the application has been active (rolled in) in the current rollin quantum (see "Scheduling Characteristics" on page 2-32 for more information). The time active is shown both in the format [[*hours* :]*minutes* :]*seconds* . *milliseconds* 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. In the example above, the partition *mypart* has a rollin quantum of one minute. The application */home/pat/glide* 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 [[*hours* :]*minutes* :]*seconds* . *milliseconds*. In the example above, the application */home/pat/glide* has been active for a total of 4 minutes and 41.60 seconds; the application *boggle* has been active for a total of 7 minutes and 12.30 seconds; and the application *myfft* has been active for a total of 2 hours, 12 minutes, and 3.90 seconds.
`COMMAND`	The command line by which the application was invoked.

Changing Partition Characteristics

Command Synopsis	Description
chpart [**-rq** *time*] [**-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.

The other partition characteristics, such as size, parent partition, and scheduling type (standard or gang), 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, are similar to the corresponding switches of **mkpart**:

-rq *time* Changes the partition's rollin quantum to *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.

-rq can be used only on a gang-scheduled partition. To use **-rq**, 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 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 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 Paragon™ OSF/1 Message-Passing System Calls 3

Introduction

Message passing is the standard means of communication among processes in Paragon OSF/1. 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 Paragon OSF/1 message-passing system calls. It 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.
- Flushing and canceling 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™ OSF/1 C System Calls Reference Manual* or the *Paragon™ OSF/1 Fortran System Calls Reference Manual* for complete information on each call.

This chapter does not describe all the Paragon OSF/1 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 Paragon OSF/1 software tools, such as TCP/IP and the X Window System, see the *Paragon™ OSF/1 Software Tools User's Guide.* For information about the system calls that require root privileges, see the *System Administrator's Guide* for your system.

Paragon OSF/1 programs written in C can also issue OSF/1 system calls. The Paragon OSF/1 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.

Paragon OSF/1 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™ OSF/1 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
setptype(*ptype*)	Set the calling process's process type.
myptype()	Obtain the calling process's process type.

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**(), and can change its process type by calling **setptype**().

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

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.

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 or changes its process type. However, a process that has changed its process type is allowed to set its process type back to a value it has used previously.

If a process changes its process type while it is running, the process type in effect when a send or receive system call was made determines the process type associated with the message. For example, suppose a process has process type 1 on node 2, but then uses **setptype**() to change its process type to 2. A message sent to process type 1 on node 2 will arrive at node 2, but will have no process to receive it (it becomes a *pending message*). If the process later uses **setptype**() to change its process type back to 1, the next time it receives a message it will get the pending message.

If a process has multiple threads of control, they may have the same or different process types. When a thread is created, it has the same process type as the thread (process) that created it. A thread can change its process type by calling **setptype()**. If two threads in a task have the same process type when a message arrives for that process type, whichever thread receives the message first gets it. See **pthread_create()** in the *OSF/1 Programmer's Reference* for information on threads.

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 call **setptype()** to set its process type to a valid value before it can send or receive any messages.

The Paragon OSF/1 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()**.

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 in Paragon OSF/1 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 specified buffer. 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™ OSF/1 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 3-20.

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 bypasses the usual flow control mechanisms; receiving with a typesel mask receives messages of a selected set of types. See the *Paragon™ OSF/1 Fortran System Calls Reference Manual* or *Paragon™ OSF/1 C System Calls Reference Manual* for information on these special message types.

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

Paragon OSF/1 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 3-22.

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 3-22 and "Extended Receive and Probe" on page 3-26).

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 3-26 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 3-13 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 3-26 for information on a version of the **irecv**() call with additional functionality.

NOTE

If a process changes its process type after making an asynchronous send or receive call, the process type that was in effect when the call was made determines the sender's process type for the message.

For example, if a process calls **irecv**() while its process type is 0, then calls **setptype**() to change its process type to 1 before the message arrives, the **irecv**() will still accept only messages for process type 0.

Releasing Message IDs

Because Paragon OSF/1 has a limited number of message IDs, you must release IDs that are no longer needed. There are four ways to release a message ID:

- You can call **msgwait**().
- You can keep calling **msgdone**() until it returns 1.
- You can call **msgignore**().
- You can call **msgcancel**(). See the section "Flushing and Canceling Messages" on page 3-17 for information on **msgcancel**().

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™ OSF/1 Fortran System Calls Reference Manual* or *Paragon™ OSF/1 C System 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 3-26 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.

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 3-26 for more information.

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 not the only way to get information about a received or pending message. See "Extended Receive and Probe" on page 3-26 for information on message-receiving and message-probing calls that also return information about the received or pending message.

Flushing and Canceling Messages

Synopsis	Description
flushmsg(*typesel, nodesel, ptypesel*)	Flush specified messages from the system.
msgcancel(*mid*)	Cancel an asynchronous send or receive operation.

Flushing Pending Messages

If after inspecting a pending message with the **info...**() calls, you decide you don't want to receive it after all, you can get rid of it with **flushmsg**(). The **flushmsg**() call clears pending messages from the system buffer. **flushmsg**() only flushes messages pending to be received (that is, waiting in a system receive buffer), not those pending to be sent (that is, waiting in a system send buffer).

For example, here is a C code fragment that checks to see if a pending message has type 1 and flushes it if it does. Otherwise, the program receives the message and continues.

```
#include <nx.h>
long buf[100], msg_type;
    •
    •
    •
cprobe(-1);
msg_type = infotype();
if (msg_type == 1) {
    flushmsg(1, mynode(), myptype());
} else {
    crecv(msg_type, buf, sizeof(buf));
}
    •
    •
    •
```

NOTE

flushmsg() can be used to flush pending messages in other processes and on other nodes.

The **flushmsg**() call can be issued by either the sender or the receiver, but the messages are flushed on the receiver. Specifically, the node number in the second parameter specifies the destination node, not the source node; and the process type in the third parameter specifies the process type of the destination process, not the source process. For example, if a process on node 1 makes the following call, it clears all pending messages of type 3 from the system buffer of the process with process type 0 on node 2:

```
/* C version */
flushmsg(3, 2, 0);

c     Fortran version
      call flushmsg(3, 2, 0)
```

All three parameters of **flushmsg**() are selection parameters. You can specify a *typesel* of -1 to flush messages of all types, a *nodesel* of -1 to flush messages on all nodes in the application, and/or a *ptypesel* of -1 to flush messages in processes of all process types on the specified node(s). For example, the following call flushes all messages waiting to be received by all processes on **mynode**():

```
/* C version */
flushmsg(-1, mynode(), -1);

c     Fortran version
      call flushmsg(-1, mynode(), -1)
```

You can also use a *typesel mask* as the *typesel* parameter, to flush messages of selected types. See the *Paragon™ OSF/1 Fortran System Calls Reference Manual* or *Paragon™ OSF/1 C System Calls Reference Manual* for information on typesel masks.

Canceling an Asynchronous Send or Receive

msgcancel() cancels an asynchronous send or receive operation. For example, assume that you post an asynchronous receive and then determine that you don't want the message. Issue a **msgcancel**(). When **msgcancel**() returns, you don't know whether the receive operation completed, but you do know that the application buffer that you set up for the received message is no longer in danger of being written into.

Consider the following situation. The sender does an asynchronous send. The receiver has not yet issued a receive call. The message goes into a system buffer on the destination node. The sender then decides that it didn't want to send the message after all. The appropriate response is for the sender to issue a **msgcancel**() followed by a **flushmsg**().

Why both? The message is either all or partially in a system buffer on the destination node. **msgcancel**() releases the sender's message ID and gets rid of any partially-sent message, but cannot affect the message if it has been completely sent; **flushmsg**() gets rid of the message if it has been completely sent, but cannot affect a message that has been only partially sent. (If the message is received before the **flushmsg**(), however, it is not affected; once received, the message is in the memory of the receiving process and cannot be affected by system calls in the sending process.)

Table 3-1 summarizes how **flushmsg()** and **msgcancel()** affect pending messages. Note that neither call affects non-pending messages (messages that have already been received).

Table 3-1. Differences Between flushmsg() and msgcancel()

Call	Affects Completely-Sent Pending Messages	Affects Partially-Sent Pending Messages	Affects Received (Non-Pending) Messages
flushmsg()	Yes	No	No
msgcancel()	No	Yes	No

Message Passing with Fortran Commons

Because Fortran does not provide structures, 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.

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.
masktrap(*state*)	Enable or disable interrupts for message handlers.
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 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 message 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, your program automatically stops what it was doing to take care of the message.

The handler function you defined 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, current processing is suspended, and **funct1()** is called with the type and length of the message and the node number and process type of the sender as arguments. The message contents can be found in the buffer specified in the **hrecv()** call (in this case, *buf1*). When **funct1()** returns, the program goes back to doing whatever it was doing when the message arrived.

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 rather than when the receive completes. **hsendrecv()** is like an **isend()** followed by an **hrecv()**, with the message ID of the **isend()** automatically released by **msgignore()**.

Passing Information to the Handler

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

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(10000);
    buf[1] = malloc(10000);
        •
        •   /* put data into the buffers */
        •
    /* send them */
    hsendx(0, buf[0], sizeof(buf[0]), 1, myptype(), freemem, 0);
    hsendx(0, buf[1], sizeof(buf[1]), 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. To prevent this, you can use the **masktrap()** call, described in the following section.

See "Extended Receive and Probe" on page 3-26 for information on a version of the **hrecv()** call with additional functionality.

Preventing Interrupts

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 **masktrap(1)** masks all your handlers. A **masktrap(0)** re-enables them. Any pending interrupts are honored when the mask is removed. For example:

```
hrecv(6,buf,sizeof(buf),myhandler);
     •
     • /* this code can be interrupted */
     • /* by a message of type 6 */
     •
masktrap(1);
     •
     • /* critical code that must not be interrupted */
     •
masktrap(0);
     •
     • /* this code can be interrupted again */
     •
```

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 6-7. 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*.

Using Other Paragon™ OSF/1 System Calls

4

Introduction

Paragon OSF/1 system calls are available to all programs running on the Intel supercomputer. These system calls provide a variety of specialized functions that let processes running on different nodes share data and coordinate their activities.

This chapter introduces the Paragon OSF/1 system calls that perform general services other than message passing. It includes the following sections, each of which describes a group of related calls:

- Controlling application execution.
- Managing partitions.
- Handling errors.
- Floating-point control.
- Miscellaneous calls.
- iPSC® 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™ OSF/1 C System Calls Reference Manual* or the *Paragon™ OSF/1 Fortran System Calls Reference Manual* for complete information on each call.

This chapter does not describe all the Paragon OSF/1 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 Paragon OSF/1 software tools, such as TCP/IP and the X Window System, see the *Paragon™ OSF/1 Software Tools User's Guide*. For information about the system calls that require root privileges, see the *System Administrator's Guide* for your system.

Paragon OSF/1 programs written in C can also issue OSF/1 system calls. The Paragon OSF/1 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.

Paragon OSF/1 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™ OSF/1 Fortran Compiler User's Guide* for information on these routines.

Controlling Application Execution

The simplest way to control the way an application executes is to use the command-line switch **-nx** when you link the application. 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 **-pn** (default *$NX_DFLT_PART*, or *.compute* if *$NX_DFLT_PART* is not set) of the size specified by **-sz** (default *$NX_DFLT_SIZE*, or all nodes of the partition if *$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).

You can perform these actions yourself, using the calls described in this section.

Controlling Application Execution with System Calls

Synopsis	Description
nx_initve(*partition, size, account, argc, argv*)	Create a new application.
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 **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. (The **...ve** suffix on some of these calls indicates that they take an argument list like that of the standard OSF/1 call **execve**().) Using these calls is more complicated than using **-nx**, but gives your program more flexibility and control.

NOTE

If you use **nx_initve()**, you should *not* link the program using **-nx**; instead, use the switch **-lnx**.

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 4-13 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 *mp_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.

nx_initve() just allocates (nonexclusively) the specified number of nodes from the partition; it does not start any processes. 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 4-14.

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 4-27). 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 the *argc* parameter is 0, as described later in this section). See "Partition Pathnames" on page 2-27 for more information on partition pathnames; see "Owner, Group, and Protection Modes" on page 2-32 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).

If the user specifies a size with the **-sz** switch on the command line, it overrides the value of the *size* parameter (unless the *argc* parameter is 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*, **-pri** *priority*, **-pn** *partition*, **-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 all nodes of 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.

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 4-14 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. See "Using PIDs" on page 4-12 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 4-12 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 (default all nodes of 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 **nx_initve()** 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 Intel 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 Paragon OSF/1 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™ OSF/1 Fortran Compiler User's Guide* for information on these routines.) Paragon OSF/1 process types are unique only within a single application and a single node; they are used with Paragon OSF/1 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);
    }
```

The Controlling Process

By calling **nx_initve**(), 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 Paragon OSF/1 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 the

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).

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, not unique to Paragon OSF/1. 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 Paragon™ OSF/1

In Paragon OSF/1, 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. When a process leaves an application, it is moved from the application's partition to the service partition, and can no longer exchange messages with the other processes in the application.

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 4-12 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 */
    nx_pri(0, 7); /* pgroup 0 specifies calling application */

    /* 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 Paragon OSF/1 message-passing 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 3-29.

- The controlling process does not participate when the application processes make any of the calls described under "Global Operations" on page 3-29.

- 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
         print *, "nx_initve failed"
         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
         print *, "nx_load failed"
         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 it 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
```

Partition Management Calls

Paragon OSF/1 provides system calls that let you create and remove partitions and change their characteristics, like the **mkpart**, **rmpart**, and **chpart** commands described in Chapter 2. See "Managing Partitions" on page 2-24 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.

To create a partition, use **nx_mkpart()**, **nx_mkpart_rect()**, or **nx_mkpart_map()**. 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).

See "Specifying the Nodes Allocated to the Partition" on page 2-38 for more information on these 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 2-27 for more information on partition pathnames; see "Owner, Group, and Protection Modes" on page 2-32 for more information on partition permissions.

size	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).
rows and *cols*	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*	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	The new partition's scheduling type: **NX_STD** to specify standard scheduling, or **NX_GANG** to specify gang scheduling. The names **NX_STD** and **NX_GANG** are defined in *nx.h* and *fnx.h*. See "Scheduling Characteristics" on page 2-32 for more information on standard and gang scheduling.

nx_mkpart(), **nx_mkpart_rect()**, and **nx_mkpart_map()** 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. All other 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 4-24.

For example, 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 gang-scheduled 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_GANG);
```

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 gang-scheduled 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_GANG);
```

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 2-27 for more information on partition pathnames; see "Owner, Group, and Protection Modes" on page 2-32 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).

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.

To change a partition's characteristics, use **nx_chpart_name**(), **nx_chpart_mod**(), **nx_chpart_epl**(), **nx_chpart_rq**(), or **nx_chpart_owner**(). 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. See "Partition Pathnames" on page 2-27 for more information on partition pathnames.

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 2-32 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 2-32 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). If it is not a multiple of 100, it is silently rounded up to the next multiple of 100. See "Scheduling Characteristics" on page 2-32 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 2-32 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.

nx_chpart_name(), **nx_chpart_mod()**, **nx_chpart_epl()**, **nx_chpart_rq()**, and **nx_chpart_owner()** 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);
```

In each of these examples, the variable *n* is assigned 0 if the call succeeded, or -1 if any error occurred.

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.

When an error occurs in a Paragon OSF/1 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 Paragon OSF/1 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. 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.)

The underscore version of a Paragon OSF/1 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);
    }
}
```

Floating-Point Control

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).

Paragon OSF/1 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.
led(*state*)	Turn the node's green LED on or off.
dclock()	Return time in seconds since booting the node.

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 Paragon OSF/1 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 2-34 for information on rollin and rollout).

Blinking the LED

The Intel supercomputer has a number of light-emitting diodes (LEDs) on its front panel that indicate the processor and message-passing status of all the nodes in the system. Among these LEDs, one green LED for each node is user-programmable. You can use the **led**() call to turn this LED on and off.

The following call turns the green LED on:

```
/* C version */
led(1);

c     Fortran version
      call led(1)
```

The following call turns the green LED off:

```
/* C version */
led(0);

c     Fortran version
      call led(0)
```

Timing Execution

dclock() returns the time in seconds since the node was last booted, as a double precision number.

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.

NOTE

Do not use **dclock()** to synchronize processes. Each node has its own counter, which differs from the counters on other nodes.

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® System Compatibility Calls

Synopsis	Description
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.
load(*filename, node, ptype*)	Load a node process.
mclock()	Return the time in milliseconds.
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*)	Return 0 (does nothing; provided for compatibility only).
waitall(*node, pid*)	Wait for node process(es) to complete.

The calls **ginv**(), **gray**(), **hwclock**(), **infopid**(), **killcube**(), **killproc**(), **load**(), **mclock**(), **mypid**(), **nodedim**(), **restrictvol**(), and **waitall**() are provided for compatibility with the iPSC series of supercomputers from Intel Corporation.

These calls should not be used in new Paragon OSF/1 programs. They either provide the same functionality as Paragon OSF/1 calls (for example, **mypid**() is identical to **myptype**() but uses the iPSC system terminology), or provide functionality that is not needed in Paragon OSF/1 (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 system, with the following exceptions:

- The only valid use of **killcube()** is **killcube(-1,-1)**.
- The only valid use of **killproc()** is **killproc(-1,-1)**.
- **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).
- 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).
- The only valid use of **waitall()** is **waitall(-1,-1)**.

See your iPSC system documentation for more information on these calls.

Using Parallel File I/O 5

Introduction

The Paragon™ OSF/1 operating system supports many of the Concurrent File System™ (CFS™) commands and calls provided by the iPSC® system. These commands and calls let you control access to files from multiple nodes.

NOTE

Only the standard, non-parallel UNIX File System (UFS) and Network File System (NFS) are currently supported. Parallel files will be provided in a future release.

The maximum length of a pathname in Paragon OSF/1 is 1024 characters; the maximum length of a single filename depends on the type of the file system containing the file (or directory). In a UFS file system, the maximum length of a filename is 255 characters; in an NFS file system, the maximum length of a filename depends on the type of the corresponding remote file system.

Increasing the Size of a File

Command Synopsis	Description
lsize [**-a**] *size file* [*file* ...]	Change the size of a file or files.

In general, you use standard OSF/1 commands such as **ls**, **cat**, **cp**, and **mv** to manipulate files in the file system. See the *OSF/1 Command Reference* for information on these commands. In addition to these commands, Paragon OSF/1 provides the **lsize** command.

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 the *file*(s) to *size* bytes.
lsize -a *size file* [*file* ...]	Increases the size of the *file*(s) 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 (it contains whatever happens to be in those disk blocks from the last time they were used).

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 Paragon OSF/1 *parallel I/O calls* you can use in parallel applications. These calls 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.

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™ OSF/1 C System Calls Reference Manual* or the *Paragon™ OSF/1 Fortran System Calls Reference Manual* for complete information on each call.

Opening Files

Before you can use a file, you must *open* it, using standard OSF/1 system calls or Fortran routines. For example, to open the file */usr/dat/mydata* for read and write access:

```
/* C version */
fd = open("/usr/dat/mydata", O_CREAT | O_RDWR, 0644);

c     Fortran version
      open(unit=10, file = '/usr/dat/mydata',
     x     status = 'new', form='unformatted')
```

NOTE

In Fortran, you must open the file with **form='unformatted'** in order to use any parallel I/O calls on the file.

See "Special Considerations for Fortran" on page 5-4 for more information.

Opening One File Per Node with "###" Filenames

If you open a file with three or more # symbols in its filename, the series of # symbols is replaced by the node number (within the application) of the node that opens the file. The node number is padded with zeros to the length of the sequence of # symbols.

For example, assume that you have the same program running on all the nodes of your application, and each node opens 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.

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.

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, opening *data.###* in every node of an application running on 2000 nodes opens files *data.000*, *data.001*, *data.002* ... *data.998*, *data.999*, *data.1000*, *data.1001* ... *data.1998*, and *data.1999*.

There is nothing special about files created in this way; each file created is a single ordinary file. For example, suppose an application creates *###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*.

Special Considerations for Fortran

This section describes the special considerations that apply when opening files in Fortran.

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.

Intel Supercomputer Systems Division recommends that you use **form='unformatted'**, and use parallel I/O calls for all file I/O. This will give you the best I/O performance.

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 can access a file in one of four I/O modes. Use **setiomode**() to change an open file's I/O mode, and **iomode**() to determine an open file's current mode.

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).

There are four 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. All files open in this mode (but you can change it with **setiomode**() after opening the file).
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.

The names **M_UNIX, M_LOG, M_SYNC,** and **M_RECORD** 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 5-31.

M_UNIX (Mode 0)

In mode **M_UNIX** (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. Because the nodes do not have to communicate (to maintain common file information), this mode offers the greatest I/O performance. This mode is the default.

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 (Mode 1)

In mode **M_LOG** (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 (Mode 2)

In mode **M_SYNC** (number 2), all nodes share a single file pointer and the nodes access the file in a synchronized round-robin fashion.

- 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 a file is a synchronizing operation, as for **M_LOG**. In addition, reading, writing, seeking (using **lseek**()) and detecting end-of-file (using **iseof**()) become synchronizing operations—they 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 3)

Mode **M_RECORD** (number 3) gives results that are similar to **M_SYNC**, but it operates more efficiently. However, **M_RECORD** requires a fixed record size.

- 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, 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.

  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.

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 Intel 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), creates a file called *mydat* in the current directory, then sends the I/O mode to the other nodes. The other nodes wait until they receive the mode, then open the file that node 0 created.

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, fail, 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, or 3):'
         read(*, 11) mode
11       format(i1)
         open(unit = nunit, file = 'mydat', iostat = fail,
     x        form = 'unformatted', status = 'new')
         if(fail .ne. 0) then
              print *, 'Could not open mydat'
              call killcube(-1, -1)
         endif
         call csend(1, mode, 4, -1, myptype())
      else
         call crecv(1, mode, 4)
         open(unit = nunit, file = 'mydat',
     x        form = 'unformatted')
      endif

      call setiomode(nunit, 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, or 3):\n");
        gets(instring);
        sscanf(instring, "%ld", &mode);
        fd = open("mydat", O_WRONLY | O_CREAT | O_TRUNC,
                  0666);
        if(fd == -1) {
            perror("mydat");
            kill(0, 9);
        }
        csend(1, &mode, sizeof(mode), -1, myptype());
    } else {
        crecv(1, &mode, sizeof(mode));
        fd = open("mydat", O_WRONLY, 0666);
    }
    setiomode(fd, mode);
    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, or 3):
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** (0), 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** (1), 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** (2), 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
```

Note that 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** (3), 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.

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 */usr/dat/mydata* and reads some data into an array called *array* using the Fortran **read** statement:

```
open(unit=10, file='/usr/dat/mydata', form='unformatted')
read 10, (array(j), j=1, n)
```

In addition to the usual I/O facilities, the Paragon OSF/1 operating system offers a series of parallel I/O calls, which are discussed in the following pages.

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 routines on the same file (for example, you must not mix **write** and **cwrite**() on the same file).

NOTE

Be careful when using parallel I/O to NFS files.

The Intel 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.

The calls **cread**() and **cwrite**() perform synchronous file I/O. They are equivalent to the standard OSF/1 calls **read**() and **write**(), except that they follow the same naming and error-handling conventions as the Paragon OSF/1 message-passing calls (see "Names of Send and Receive Calls" on page 3-7 for information on the Paragon OSF/1 system call naming conventions; see "Handling Errors" on page 4-27 for information on the Paragon OSF/1 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 */usr/dat/hello*:

```
fd = open("/usr/dat/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 */usr/dat/mydata*, 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** (the default).

```
      open(unit=10, file='/usr/dat/mydata', 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**(). 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 you need to detect errors in reading and writing, you must program in C and use either the standard OSF/1 calls (**read**() and **write**(), described in the *OSF/1 Programmer's Reference*) or the underscore versions of the parallel I/O calls (**_cread**() and **_cwrite**(), described under "Handling Errors" on page 4-27).

Asynchronous File I/O

Synopsis	Description
iread(*fileID*, *buffer*, *nbytes*)	Asynchronous read from a file. (Do not wait for completion.)
iwrite(*fileID*, *buffer*, *nbytes*)	Asynchronous write to a file. (Do not wait 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.

The calls **iread**() and **iwrite**() perform asynchronous file I/O.

NOTE

The asynchronous calls currently work the same as the synchronous calls; they do not return until the specified I/O operation has completed. True asynchronous operation will be provided in a future release.

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.

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)
```

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.

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 "Floating-Point Control" on page 4-29) 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 is some buffering within the operating system, which cannot be avoided.)

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 (it contains whatever happens to be in those disk blocks from the last time they were used).

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.

Performing Extended Arithmetic

Synopsis		Description
eadd(*e1*, *e2*)	(C)	Add two extended numbers.
eadd(*e1*, *e2*, *eresult*)	(Fortran)	
ecmp(*e1*, *e2*)		Compare two extended numbers.
ediv(*e*, *n*)	(C)	Divide extended number by integer.
ediv(*e*, *n*, *result*)	(Fortran)	
emod(*e*, *n*)	(C)	Give extended number modulo an integer (remainder when *e* is divided by *n*).
emod(*e*, *n*, *result*)	(Fortran)	
emul(*e*, *n*)	(C)	Multiply extended number by integer.
emul(*e*, *n*, *eresult*)	(Fortran)	
esub(*e1*, *e2*)	(C)	Subtract two extended numbers.
esub(*e1*, *e2*, *eresult*)	(Fortran)	
etos(*e*, *s*)		Convert extended number to string.
stoe(*s*)	(C)	Convert string to extended number.
stoe(*s*, *e*)	(Fortran)	

The extended arithmetic calls manipulate 64-bit integers, also called *extended numbers*. You use these calls to manipulate the parameters used by some parallel I/O calls (described in the following section).

Extended numbers are unsigned 64-bit integers with values from 0 to $2^{64} - 1$ (approximately 1.8×10^{19}).

- In Fortran, extended numbers are stored in a two-element array of type **integer*4**.
- In C, extended numbers 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 number, rather than access its internal structure.

Some of these calls return extended numbers. 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 numbers *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 number, you must create your own extended number from the desired integer value. To create an extended number, use **stoe**(). This call takes a string whose value is a number, and returns the corresponding numeric value as an extended number. For example, the following code fragment adds 1 to the value of the extended number *e1*. It does this by converting the string "1" to an extended number with **stoe**(), storing the resulting extended number 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 numbers. When you use **ediv**() or **emod**(), the divisor and answer must be 4-byte integers, not extended numbers. Similarly, when you use **emul**(), the second argument must be a 4-byte integer, not an extended number.

You can also compare two extended numbers; **ecmp**() returns -1, 0, or 1, depending on whether the first extended number is less than, equal to, or greater than the second.

Extended File Manipulation Calls

Synopsis		Description
eseek(*fildes, offset, whence*)	(C)	Move file pointer in extended file.
eseek(*unit, offset, whence, newpos*)	(Fortran)	
esize(*fildes, offset, whence*)	(C)	Increase size of extended file.
esize(*unit, offset, whence, newsize*)	(Fortran)	
estat(*path, buffer*)	(C only)	Get status of extended file from pathname.
festat(*fildes, buffer*)	(C only)	Get status of open extended file from file descriptor.

The **e...()** calls perform standard OSF/1 file operations, but have parameters that are *extended numbers* (a data type capable of representing integers greater than 2G – 1). You must use the calls described under "Performing Extended Arithmetic" on page 5-23 to operate on extended numbers.

- The call **eseek()** is like **lseek()** (discussed under "Detecting End-of-File and Moving the File Pointer" on page 5-19), except that the *offset* parameter is an extended number. The C version of this call is a function that returns the new position as an extended number; 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 5-21), except that the *offset* parameter is an extended number. The C version of this call is a function that returns the new size as an extended number; the Fortran version is a subroutine that stores the new size in its fourth parameter.

- The calls **estat()** and **festat()** are like the standard OSF/1 calls **stat()** 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 number. These calls are available only in C, not in Fortran.

NOTE

Although Paragon OSF/1 provides these calls for compatibility with the iPSC system, it does not currently support files larger than 2G – 1 bytes in size. Files larger than 2G – 1 bytes will be supported in a future release.

Closing Files in Parallel

Use the standard OSF/1 system calls or Fortran routines to close files. 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 5-6 for more information.

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).

<*sys/mtio.h*> defines two constants you can use as the second argument of **ioctl**():

MTIOCTOP	Perform operation on tape.
MTIOCGET	Get status of tape.

The rest of this section explains the details of using these constants.

Naming Tape Devices

The Paragon OSF/1 operating system uses the following conventions for naming tape devices:

/dev/tapeN	Cartridge tape, rewinds automatically when closed.
/dev/ntapeN	Cartridge tape, does not rewind automatically when closed.
/dev/rtapeN	Raw cartridge tape, rewinds automatically when closed.
/dev/nrtapeN	Raw cartridge tape, does not rewind automatically when closed.
/dev/3480tapeN	3480 tape, rewinds automatically when closed.
/dev/n3480tapeN	3480 tape, does not rewind automatically when closed.
/dev/r3480tapeN	Raw 3480 tape, rewinds automatically when closed.
/dev/nr3480tapeN	Raw 3480 tape, does not rewind automatically when closed.

In each case, N is the number of the corresponding physical tape device. The first device of each type on the system is number 0, the second is number 1, and so on. So, for example, to use the first cartridge tape device with normal (block buffered) access and have it rewind automatically when closed, use the pathname */dev/tape0*. To use the same device with raw (unbuffered) access and have it *not* rewind automatically when closed, use the pathname */dev/nrtape0*.

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. The *mtop* structure is defined as follows:

```
struct mtop {
    short  mt_op;     /* operation to perform */
    short  fill;      /* ignored */
    long   mt_count; /* how many operations to perform */
};
```

This structure tells **ioctl()** what operation to perform. The valid values of the *mt_op* field 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.
MTNOP	No operation, sets status only.
MTRETEN	Retension the tape.
MTERASE	Erase the entire tape.
MTEOM	Position the tape at end of media (SCSI 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/tape0*:

```
#include <fcntl.h>
#include <errno.h>
#include <sys/mtio.h>

main() {
    int fd;
    struct mtop s;

    fd = open("/dev/tape0", O_RDONLY, 0666);
    if(fd == -1) {
        perror("opening /dev/tape0");
        exit(1);
    }

    s.mt_op = MTREW;
    s.mt_count = 1;
    if (ioctl(fd, MTIOCTOP, &s) == -1) {
        perror("rewinding tape");
        exit(2);
    }
}
```

Getting Status of Tape Devices

When you call **ioctl()** with **MTIOCGET** 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 */
};
```

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.

For example, the following C program prints the status of the device connected to */dev/tape0*:

```
#include <fcntl.h>
#include <errno.h>
#include <sys/mtio.h>

main() {
    int fd;
    struct mtget s;

    fd = open("/dev/tape0", O_RDONLY, 0666);
    if(fd == -1) {
        perror("opening /dev/tape0");
        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);
}
```

Synchronization Summary

Table 5-1 lists the I/O modes and summarizes the I/O calls that are synchronizing calls in each one. Table 5-2 lists the most commonly-used I/O calls and summarizes the I/O modes that cause them to become synchronizing calls.

Table 5-1. Synchronization in Each I/O Mode

I/O Mode	I/O Calls that Synchronize
M_UNIX	setiomode()
M_LOG	setiomode() and close()
M_SYNC	All
M_RECORD	setiomode(), lseek(), eseek(), and close()

Table 5-2. File I/O Calls that Synchronize

Call	I/O Modes Causing the Call to Synchronize
close()	M_LOG, M_SYNC, and M_RECORD
cread()	M_SYNC
cwrite()	M_SYNC
eseek()	M_SYNC and M_RECORD
iread()	M_SYNC
iseof()	M_SYNC
iwrite()	M_SYNC
lseek()	M_SYNC and M_RECORD
setiomode()	All

Designing a Parallel Application 6

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.

The Paragon™ OSF/1 Programming Model

As described in Chapter 1, the Intel 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 4-13 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*.

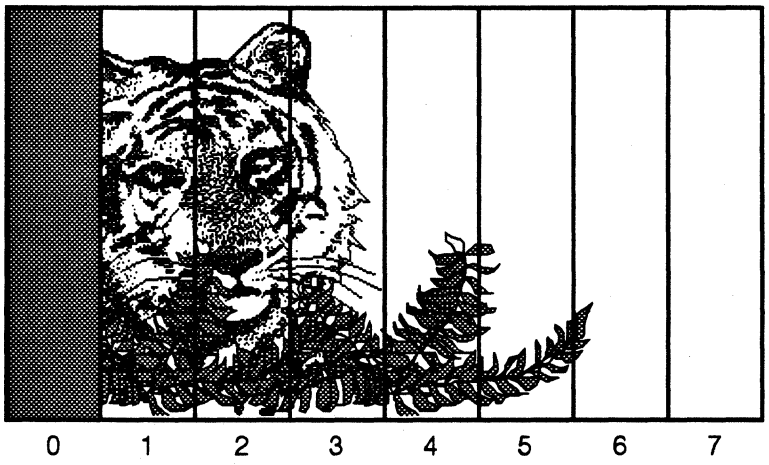
Domain Decomposition

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 6-1). Because some parts of the image may be more detailed than others, they will require more processing. The shaded portion of Figure 6-1 shows the work done by node 0. If you divide the image sequentially among the nodes, as shown in the top half of Figure 6-1, 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 6-1, 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 6-1. 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 6-1 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.

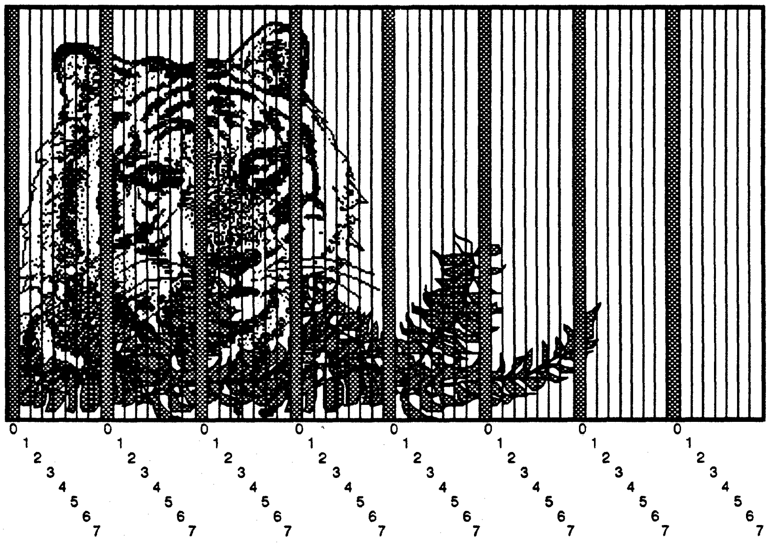


Figure 6-1. Using Domain Decomposition to Achieve Load Balancing

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 3-29, 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 Intel 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 an Intel 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 Intel 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 6-2. 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.)

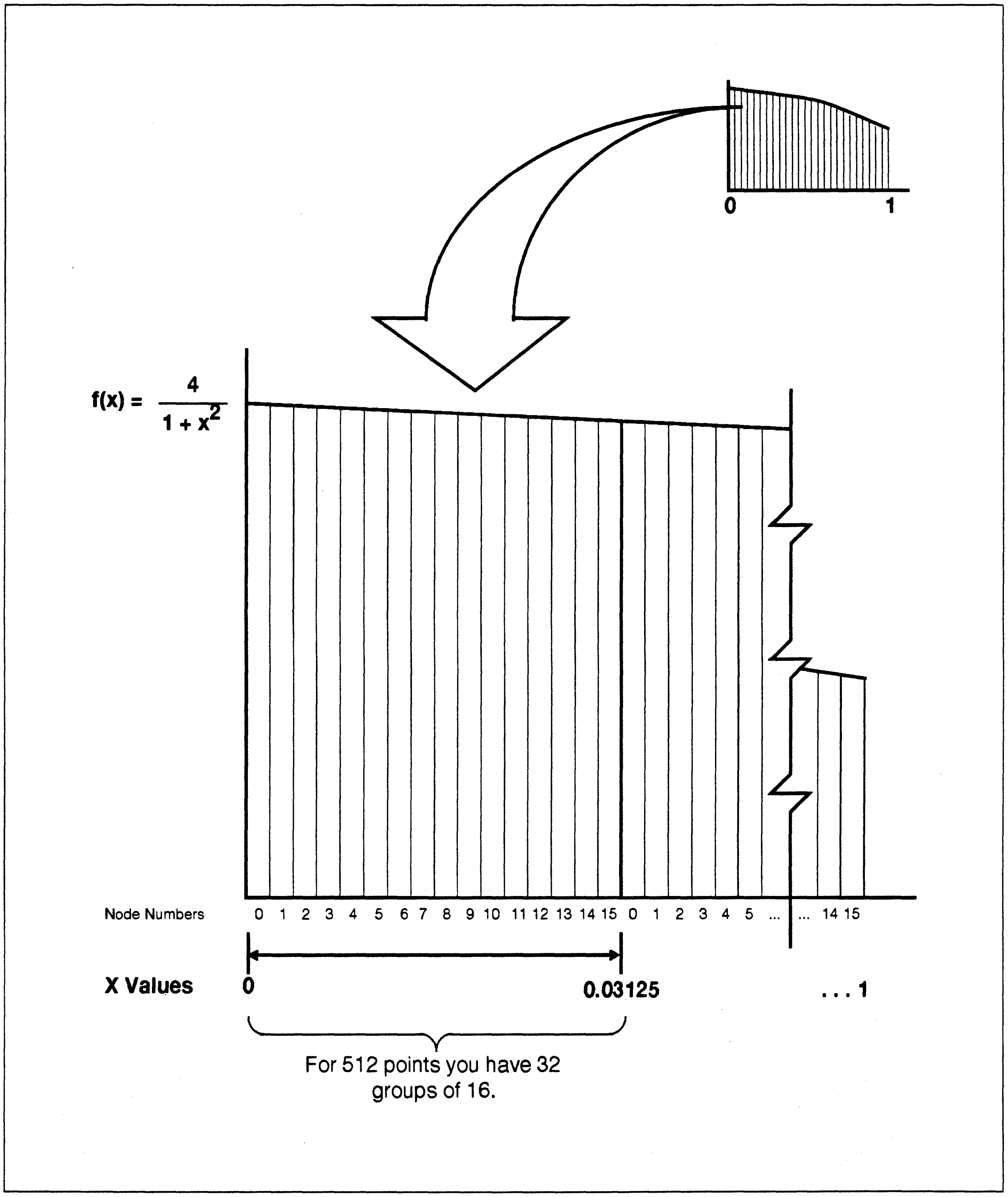


Figure 6-2. The Decomposition Used for the pi Example

Here is the parallel version of the program. The source for this program is available on the Intel 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™ OSF/1 Fortran System 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 an Intel 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 6-3 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 6-3, 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 6-3, 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 6-3, 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.

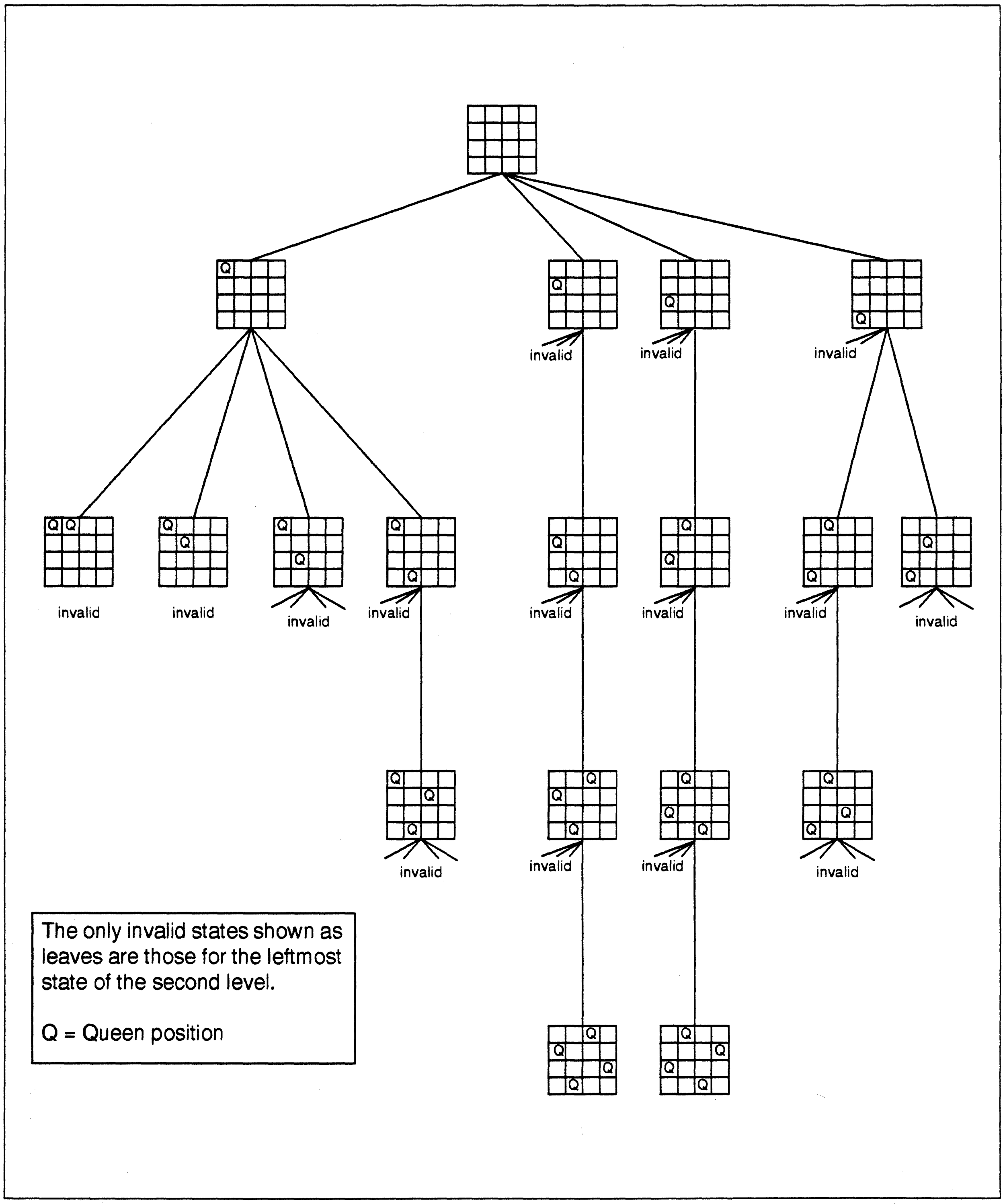
Manager/worker decomposition (a type of control decomposition) is a useful way of achieving balanced computational loads when the application consists of a large number of tasks that are of varying length. Because there is no way of determining up front what the length of the task is, the method consists 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 6-3. 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.

Summary of Commands and System Calls

A

This appendix summarizes the commands and system calls of Paragon™ OSF/1. 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 Paragon OSF/1. 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 Chapters 2 and 5. See the *Paragon™ OSF/1 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 a Paragon OSF/1 application written in C on an Intel supercomputer.
f77 -nx [*switches*] *sourcefile...*	Compile a Paragon OSF/1 application written in Fortran on an Intel supercomputer.
icc -nx [*switches*] *sourcefile...*	Compile a Paragon OSF/1 application written in C on an Intel supercomputer or cross-development workstation.
if77 -nx [*switches*] *sourcefile...*	Compile a Paragon OSF/1 application written in Fortran on an Intel supercomputer or cross-development workstation.

Running Applications

Table A-2. Commands for Running Applications

Command Synopsis	Description
application [**-sz** *size*] [**-pri** *priority*] [**-pt** *ptype*] [**-on** *nodespec*] [**-pn** *partition*] [**-mbf** *memory_buffer*] [**-mex** *memory_export*] [**-mea** *memory_each*] [**-pkt** *packet_size*] [**-sth** *send_threshold*][**-sct** *send_count*] [**-gth** *give_threshold*] [**-plk**] [*application_args*] [**\;** *file* [**-pt** *ptype*] [**-on** *nodespec*] [*application_args*]] ...	Execute a Paragon OSF/1 application.

Managing Partitions

Table A-3. Commands for Managing Partitions

Command Synopsis	Description
mkpart [**-sz** *size* | **-sz** *hXw* | **-nd** *nodespec*] [**-ss** | [[**-rq** *time*] [**-epl** *priority*]]] [**-mod** *mode*] *partition*	Create a partition.
rmpart [**-f**] [**-r**] *partition*	Remove a partition.
showpart [*partition*]	Show the characteristics of a partition.
lspart [**-r**] [*partition*]	List the subpartitions of a partition.
pspart [*partition*]	List the applications in a partition.
chpart [**-epl** *priority*] [**-g** *group*] [**-mod** *mode*] [**-nm** *name*] [**-o** *owner*[**.** *group*]] [**-rq** *time*] *partition*	Change certain partition characteristics.

Increasing the Size of a File

Table A-4. Commands for Increasing the Size of a File

Command Synopsis	Description
lsize [**-a**] *size file* ...	Change the size of a file or 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
fsplit [*filename*]	Split one file containing several Fortran program units into several files containing one program unit each. (See the *Paragon™ OSF/1 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™ OSF/1 Software 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 *System Administrator's Guide* for your system for more information.)

C System Call Summary

This section summarizes the C versions of the system calls discussed in Chapters 3, 4, and 5. See the *Paragon™ OSF/1 C System 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.
void **setptype**(long *ptype*);	Set the calling process's process type.
long **myptype**(void);	Obtain the calling process's process type.
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.

Flushing and Canceling Messages

Table A-11. C Calls for Flushing and Canceling Messages

Synopsis	Description
void **flushmsg**(long *typesel*, long *node*sel, long *ptypesel*);	Flush specified messages from the system.
void **msgcancel**(long *mid*);	Cancel an asynchronous send or receive operation.

Treating a Message as an Interrupt

Table A-12. 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.
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 reply arrives.

Extended Receive and Probe

Table A-13. 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-14. 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-14. 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-14. 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-15. C Calls for Controlling Application Execution

Synopsis	Description
long **nx_initve**(char **partition*, long *size*, char **account*, long **argc*, char **argv*[ ]);	Create a new application.
long **nx_pri**(long *pgroup*, long *priority*);	Set the priority of an application.
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.

Partition Management

Table A-16. C Calls for Partition Management

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_rmpart**(char **partition*, long *force*, long *recursive*);	Remove a partition.
long **nx_chpart_name**(char **partition*, char **name*);	Change a partition's name.
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.

Handling Errors

Table A-17. 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-18. 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-19. 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 node.

iPSC® System Compatibility

Table A-20. C Calls for iPSC® System Compatibility (1 of 2)

Synopsis	Description
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.
unsigned long **mclock**(void);	Return the time in milliseconds.
long **mypid**(void);	Return the process type of the calling process.

Table A-20. C Calls for iPSC® System Compatibility (2 of 2)

Synopsis	Description
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).
void **waitall**(long *node*, long *pid*);	Wait for node process(es) to complete.

I/O Modes

Table A-21. C Calls for I/O Modes

Synopsis	Description
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-22. 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.
long **iread**(int *fildes*, char **buffer*, unsigned int *nbytes*);	Asynchronous read from a file. (Do not wait for completion.)
long **iwrite**(int *fildes*, char **buffer*, unsigned int *nbytes*);	Asynchronous write to a file. (Do not wait 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-23. 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-24. 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-25. 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 **festat**(int *fildes*, struct estat **buffer*);	Get status of open extended file from file descriptor or unit.

Performing Extended Arithmetic

Table A-26. C Calls for Performing Extended Arithmetic

Synopsis	Description
esize_t **eadd**(esize_t *e1*, esize_t *e2*);	Add two extended numbers.
long **ecmp**(esize_t *e1*, esize_t *e2*);	Compare two extended numbers.
long **ediv**(esize_t *e*, long *n*);	Divide extended number by integer.
long **emod**(esize_t *e*, long *n*);	Give extended number modulo an integer (remainder when *e* is divided by *n*).
esize_t **emul**(esize_t *e*, long *n*);	Multiply extended number by integer.
esize_t **esub**(esize_t *e1*, esize_t *e2*);	Subtract two extended numbers.
void **etos**(esize_t *e*, char **s*);	Convert extended number to string.
esize_t **stoe**(char **s*);	Convert string to extended number.

Fortran System Call Summary

This section summarizes the Fortran versions of the system calls discussed in Chapters 3, 4, and 5. See the *Paragon™ OSF/1 Fortran System Calls Reference Manual* for more information on these calls.

Process Characteristics

Table A-27. 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.
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-28. 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-29. 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-29. 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-30. 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-31. 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.

Flushing and Canceling Messages

Table A-32. Fortran Calls for Flushing and Canceling Messages

Synopsis	Description
SUBROUTINE **FLUSHMSG**(*typesel*, *node*sel, *ptypesel*) INTEGER *typesel* INTEGER *node*sel INTEGER *ptypesel*	Flush specified messages from the system.
SUBROUTINE **MSGCANCEL**(*mid*) INTEGER *mid*	Cancel an asynchronous send or receive operation.

Treating a Message as an Interrupt

Table A-33. 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.

Table A-33. Fortran Calls for Treating a Message as an Interrupt (2 of 2)

Synopsis	Description
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.
INTEGER FUNCTION **MASKTRAP**(*state*) INTEGER *state*	Enable or disable interrupts for message handlers.
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 reply arrives.

Extended Receive and Probe

Table A-34. 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.
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.

Table A-34. Fortran Calls for Extended Receive and Probe (2 of 2)

Synopsis	Description
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-35. 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-35. 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-35. 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-36. 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_PRI**(*pgroup, priority*) INTEGER *pgroup* INTEGER *priority*	Set the priority of an application.
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.

Table A-36. Fortran Calls for Controlling Application Execution (2 of 2)

Synopsis	Description
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.

Partition Management

Table A-37. Fortran Calls for Partition Management (1 of 2)

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.
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.

Table A-37. Fortran Calls for Partition Management (2 of 2)

Synopsis	Description
INTEGER FUNCTION **NX_RMPART**(*pathname*, *force*, *recursive*) CHARACTER *partition**(*) INTEGER *force* INTEGER *recursive*	Remove a partition.
INTEGER FUNCTION **NX_CHPART_NAME**(*partition*, *name*) CHARACTER *partition**(*) CHARACTER *name**(*)	Change a partition's name.
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.

Handling Errors

Table A-38. 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-39. 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-40. 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 node.

iPSC® System Compatibility

Table A-41. Fortran Calls for iPSC® System Compatibility (1 of 2)

Synopsis	Description
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).
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.
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 $2^{dimension}$).

Table A-41. Fortran Calls for iPSC® System Compatibility (2 of 2)

Synopsis	Description
INTEGER FUNCTION **RESTRICTVOL**(*unit*, *nvol*, *vollist*) INTEGER *unit* INTEGER *nvol* INTEGER *vollist*(*)	Return 0 (does nothing; provided for compatibility only).
SUBROUTINE **WAITALL**(*node*, *pid*) INTEGER *node* INTEGER *pid*	Wait for node process(es) to complete.

I/O Modes

Table A-42. Fortran Calls for I/O Modes

Synopsis	Description
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-43. Fortran Calls for Reading and Writing Files in Parallel

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.
INTEGER FUNCTION **IREAD**(*unit, buffer, nbytes*) INTEGER *unit* INTEGER *buffer(*)* INTEGER *nbytes*	Asynchronous read from a file. (Do not wait for completion.)
INTEGER FUNCTION **IWRITE**(*unit, buffer, nbytes*) INTEGER *unit* INTEGER *buffer(*)* INTEGER *nbytes*	Asynchronous write to a file. (Do not wait 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-44. 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-45. 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-46. 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-47. 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-48. 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 numbers.
INTEGER FUNCTION **ECMP**(*e1*, *e2*) INTEGER *e1*(2) INTEGER *e2*(2)	Compare two extended numbers.
SUBROUTINE **EDIV**(*e*, *n*, *result*) INTEGER *e*(2) INTEGER *n* INTEGER *result*	Divide extended number by integer.
SUBROUTINE **EMOD**(*e*, *n*, *result*) INTEGER *e*(2) INTEGER *n* INTEGER *result*	Give extended number 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 number by integer.
SUBROUTINE **ESUB**(*e1*, *e2*, *eresult*) INTEGER *e1*(2) INTEGER *e2*(2) INTEGER *eresult*(2)	Subtract two extended numbers.
SUBROUTINE **ETOS**(*e*, *s*) INTEGER *e*(2) CHARACTER *s*(*)	Convert extended number to string.
SUBROUTINE **STOE**(*s*, *e*) CHARACTER *s*(*) INTEGER *e*(2)	Convert string to extended number.

iPSC® System Compatibility B

Introduction

This appendix gives you information you can use to port programs to Paragon™ OSF/1 from the iPSC® series of supercomputers from Intel Supercomputer Systems Division.

This appendix lists the differences between iPSC system commands and system calls and those of Paragon™ OSF/1, 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 Paragon OSF/1 as they do in the iPSC system.

NOTE

There is no longer an SRM. All software development is done either on remote workstations or on the Intel supercomputer itself. Parallel applications are run only on the Intel supercomputer.

New Features

Paragon OSF/1 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 Intel supercomputer, 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 "Specifying Message-Passing Configuration Parameters" on page 2-21.
- 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 2-24 and "Partition Management Calls" on page 4-20.
- 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.
- You can pass a parameter to a message interrupt handler with **hsendx**(), as discussed under "Treating a Message as an Interrupt" on page 3-22.
- 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 3-26.
- You can use system calls to control the execution characteristics of parallel programs, as discussed under "Controlling Application Execution" on page 4-2.

Compilers

The Paragon OSF/1 compilers work the same as the iPSC system compilers, with the following exceptions:

- The compilers, linker, and other tools are now available on the Intel supercomputer 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™ OSF/1 Software 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™ OSF/1 Fortran Compiler User's Guide* or *Paragon™ OSF/1 C Compiler User's Guide* for more information on the Paragon OSF/1 compilers.

NOTE

You cannot use the Paragon OSF/1 cross-compilers to produce programs for the iPSC system, and you cannot use the iPSC system cross-compilers to produce programs for Paragon OSF/1.

If you develop programs for the iPSC system as well for Paragon OSF/1, 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 Paragon OSF/1, 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 Paragon OSF/1, but none of the iPSC-system-specific commands are supported. However, many of these commands are not needed in Paragon OSF/1, or have equivalent standard commands in OSF/1.

Cube Control Commands

The usage model of Paragon OSF/1 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 Paragon OSF/1 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 Paragon OSF/1.)

For this reason, the following iPSC system commands, which create and control cubes, are not supported in Paragon OSF/1:

archcube	This command is not needed in Paragon OSF/1 because all nodes currently have the same architecture.
attachcube	This command is not needed in Paragon OSF/1 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 2-45 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 2-14 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 2-38 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 2-23 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 Paragon OSF/1 because you can use the usual OSF/1 I/O redirection characters to redirect an application's output. See "I/O Redirection" on page 2-11 for more information.
relcube	This command is not needed in Paragon OSF/1 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 2-42 for more information.

startcube	This command has no equivalent in Paragon OSF/1. There is no way to load an application into the nodes' memory without starting it.
syslog	This command is not needed in Paragon OSF/1 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 2-11 for more information.
waitcube	This command is not needed in Paragon OSF/1 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 2-11 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 Paragon OSF/1:

cptape	Use the **cpio** command instead. See **cpio** in the *OSF/1 Command Reference* for more information.
showvol	Use the **mount** command to get a listing of the currently-mounted file systems. See **mount** in the *OSF/1 Command Reference* for more information.
star	Use the **tar** command instead. See **tar** in the *OSF/1 Command Reference* for more information.
stream	This command is not needed in Paragon OSF/1 because there is no streaming tape drive.
tapemode	This command currently has no equivalent in Paragon OSF/1. 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 Paragon OSF/1:

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 Paragon OSF/1. 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 Paragon OSF/1:

rf77	Use the **if77** command instead. See the *Paragon™ OSF/1 Fortran Compiler User's Guide* for more information.
rcc	Use the **icc** command instead. See the *Paragon™ OSF/1 C Compiler User's Guide* for more information.
rld	Use the **ld860** command instead. See the *Paragon™ OSF/1 Fortran Compiler User's Guide* or *Paragon™ OSF/1 C Compiler User's Guide* for more information.

ras	Use the **as860** command instead. See the *Paragon™ OSF/1 Fortran Compiler User's Guide* or *Paragon™ OSF/1 C Compiler User's Guide* for more information.
rar	Use the **ar860** command instead. See the *Paragon™ OSF/1 Fortran Compiler User's Guide* or *Paragon™ OSF/1 C Compiler User's Guide* for more information.

Miscellaneous Commands

The following iPSC system commands are not supported in Paragon OSF/1:

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 Intel supercomputer from your workstation. See **rlogin** or **telnet** in your workstation's documentation for more information.
rebootcube	This command has no equivalent in Paragon OSF/1. 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 Paragon OSF/1. 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 *Paragon™ OSF/1 C System Calls Reference Manual* or *Paragon™ OSF/1 Fortran System Calls Reference Manual*; see "iPSC® System Compatibility Calls" on page 4-36 for a list of these calls.

Include Files

Paragon OSF/1 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 Paragon OSF/1 do not usually have host programs. The usual programming model in Paragon OSF/1 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 "Controlling Application Execution" 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 Paragon OSF/1:

attachcube() This call currently has no equivalent in Paragon OSF/1. 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 "Controlling Application Execution" on page 4-2 for information on creating and controlling applications using system calls.

cubeinfo() This call currently has no equivalent in Paragon OSF/1. However, because allocation of nodes in Paragon OSF/1 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 **nx_initve()** instead. See "Controlling Application Execution" on page 4-2 for information on **nx_initve()**.

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, and then use **flushmsg()** to flush messages related to that process. See "Flushing and Canceling Messages" on page 3-17 for information on **flushmsg()**.

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 4-12.

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 Paragon OSF/1 are equivalent to NX PIDs in the iPSC system. PIDs (*pid_array* in this example) in Paragon OSF/1 are standard UNIX process IDs.

See the *OSF/1 Programmer's Reference* for information on **kill()**; see "Controlling Application Execution" 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 Paragon OSF/1. 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 Paragon OSF/1. 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 4-17 for information on using **setptype()** in a controlling process.

setsyslog()
This call is not necessary in Paragon OSF/1. 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()
This call is supported, but can only be used to wait for all processes on all nodes (**waitall(-1,-1)**).

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 4-12.

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 Paragon OSF/1 are equivalent to NX PIDs in the iPSC system. PIDs (*pid_array* in this example) in Paragon OSF/1 are standard UNIX process IDs.

See the *OSF/1 Programmer's Reference* for information on **wait()** and **waitpid()**; see "Controlling Application Execution" 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 "Controlling Application Execution" 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 Paragon OSF/1.

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 Paragon OSF/1 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 Paragon OSF/1 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 "Floating-Point Control" on page 4-29 for the correct values for Paragon OSF/1.

Miscellaneous Calls

The following iPSC system calls are not supported in Paragon OSF/1:

getiphosts()	This call currently has no equivalent in Paragon OSF/1. 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 Paragon OSF/1. 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 Paragon OSF/1. 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™ OSF/1 Fortran Compiler User's Guide* for information on the Fortran interface to **signal**().

plogon() and plogoff()

These calls currently have no equivalent in Paragon OSF/1. 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 Paragon OSF/1. 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.

Summary

Table B-2 summarizes the Paragon OSF/1 equivalents for the unsupported iPSC system commands.

Table B-2. Summary of Unsupported iPSC® System Commands (1 of 2)

iPSC® System Command	Paragon™ OSF/1 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**
plogoff	(none)

Table B-2. Summary of Unsupported iPSC® System Commands (2 of 2)

iPSC® System Command	Paragon™ OSF/1 Equivalent
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 Paragon OSF/1 equivalents for the unsupported iPSC system calls.

Table B-3. Summary of Unsupported iPSC® System Calls

iPSC® System Call	Paragon™ OSF/1 Equivalent
attachcube()	(none)
cubeinfo()	(none)
fpgetsticky(), **fpsetsticky()**, **fpgetmask()**, **fpsetmask()**	Supported, except for **FP_X_DNML**, but Fortran mask values are different.
getcube()	**nx_initve()**
getiphosts()	(none)
gixor()	(none)
glxor()	(none)
handler()	**signal()**
killcube()	Use **killcube(-1,-1)** to kill and flush all processes; use **kill()** followed by **flushmsg()** to kill and flush one process
killsyslog()	(none)
killproc()	Use **killproc(-1,-1)** to kill all processes; use **kill()** to kill one process
newserver()	**freopen()**
plogoff()	(none)
plogon()	(none)
relcube()	(none)
setiphost()	(none)
setpid()	**setptype()**
setsyslog()	(none)
waitall()	Use **waitall(-1,-1)** 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