Using Parallel File I/O 5

Introduction

The Paragon™ OSF/1 operating system provides two forms of parallel I/O to files:

- A special file system type called *PFS*, for *Parallel File System*, gives applications high-speed access to a large amount of disk storage. PFS file systems are optimized for simultaneous access by multiple nodes. Files in PFS file systems can be very large (up to several terabytes); the exact maximum depends on your system configuration. Access to PFS file systems also uses an I/O technique called *fast path I/O*, which gives superior performance for large I/O operations (64K bytes or more per read or write).

- Special I/O system calls, called *parallel I/O calls*, facilitate I/O from multiple nodes and permit I/O to very large files in PFS file systems. These calls can give applications better performance and more control over parallel file I/O than is offered by the standard C and Fortran file I/O features. These calls are compatible with the Concurrent File System™ (CFS™) calls provided by the iPSC® system.

A system running Paragon OSF/1 can have both PFS and non-PFS file systems. You can access files in PFS file systems with both parallel I/O calls and non-parallel I/O calls; you can use parallel I/O calls to access files in both PFS file systems and non-PFS file systems. However, in most cases you get the best performance when you use parallel I/O calls to access files in PFS file systems.

This chapter discusses both PFS file systems and parallel I/O calls. It also gives information on performing operations on tape devices in Paragon OSF/1. For information on getting the best performance from PFS file systems and parallel I/O calls, see “I/O Performance” on page 8-23.

Disks and File Systems

Every Intel supercomputer has one or more *disk devices* attached to it. Each disk device is either a single hard disk or a *RAID subsystem*. RAID stands for Redundant Array of Inexpensive Disks; in a RAID subsystem, several hard disks are connected together into a unit that appears to the system as a single large disk drive. Files stored to a RAID subsystem are distributed, or *striped*, among the disks within it by the RAID controller hardware.

Each disk device is controlled by an *I/O node*: a compute node with an I/O connection. I/O nodes communicate with the other nodes in the system using the node-to-node message-passing network and with the disk drives using a SCSI interface (or other interface). The I/O nodes may or may not also run application processes; this is determined by your system administrator. Each I/O node can control up to seven disk devices, and the number of I/O nodes is limited only by the number of slots in the system, so the total amount of disk space that could be installed in an Intel supercomputer is a terabyte or more.

The set of disk devices connected to the Intel supercomputer's I/O nodes is divided into *file systems.* A file system can encompass anything from a portion of the space on one disk device to all of the space on several disk devices. A file system is made accessible by *mounting* it to a directory (this requires system administrator privileges). This directory is called the file system's *mount point*. For example, if the file system */dev/io0/rz0f* is mounted on the directory */home* (the directory */home* is the file system's mount point), whenever you write a file in */home* it is stored in the file system */dev/io0/rz0f*.

Each file system has a *type* that describes its internal structure and determines some of the operations that can be performed on it. The supported file system types are:

UFS	UNIX File System, the standard file system type for OSF/1.
NFS	Network File System, a file system type that represents a file system on another computer on the network.
PFS	Parallel File System, a file system type that is optimized for access by parallel processes. This file system type is unique to Paragon OSF/1.

This chapter discusses how PFS file systems work and how you can use the parallel I/O system calls provided by Paragon OSF/1 to access files in file systems of all types.

PFS File Systems and PFS Files

Internally, a file system of type PFS consists of one or more *stripe directories*. The stripe directories that make up a PFS file system are determined by the system administrator when the PFS file system is mounted.

Each stripe directory is usually the mount point of a separate UFS file system. Just as a RAID subsystem collects together several hard disks into a unit that behaves like a single large disk, a PFS file system collects together several file systems into a unit that behaves like a single large file system. A system running Paragon OSF/1 can have any number of PFS file systems.

The maximum storage capacity of a PFS file system is the sum of the capacities of the different file systems containing its stripe directories. For example, if a PFS file system consists of four stripe directories, each of which is the mount point of a UFS file system with a capacity of 100M bytes, the capacity of the PFS file system is 400M bytes. However, if another PFS file system also consists of four stripe directories, but two of them are directories in one UFS file system with a capacity of 100M bytes and the other two are directories in another UFS file system with a capacity of 100M bytes, the capacity of the PFS file system is only 200M bytes.

A *PFS file* is any ordinary file that is stored in a file system of type PFS. PFS files are distributed, or *striped*, across the stripe directories that make up the PFS file system. The amount of data from a PFS file that is stored in each stripe directory is determined by the PFS file system's *stripe unit*, a quantity that is set by the system administrator when the PFS file system is mounted. The maximum size of a file in a PFS file system is roughly 2G bytes times the number of file systems in the PFS file system.[1]

For example, suppose a PFS file system consists of four stripe directories and has a stripe unit of 4K bytes. When you write a 20K-byte file to this PFS file system, the first 4K bytes of the file are stored in the first stripe directory, the second 4K bytes in the second stripe directory, the third 4K bytes in the third stripe directory, the fourth 4K bytes in the fourth stripe directory, and the last 4K bytes back in the first stripe directory.

Objects in PFS file systems that are not ordinary files (objects such as directories, symbolic links, and device special files) are not striped; each such object exists on just one disk.

PFS Filenames and Pathnames

Filenames and pathnames in PFS file systems work the same as pathnames in UFS file systems. The maximum length of a pathname is 1024 characters; the maximum length of a single filename is 255 characters.

---

1. The exact maximum size is given by the formula $((((2G - 1) - r) \times n) + r)$, where r is $(2G - 1)$ **mod** *stripe_unit* (that is, the remainder when the largest integer multiple of the stripe unit that is less than 2G – 1 is subtracted from 2G – 1) and n is the number of *different* file systems containing the PFS file system's stripe directories.

PFS Limitations

In the current release, PFS file systems and parallel I/O calls have the following limitations:

- PFS files cannot be accessed from a remote system via NFS.
- PFS does not support executable files. If you copy a binary file to a PFS file system and try to execute it, an "Operation not supported by this file system" error occurs.
- PFS does not support core files. If a core dump occurs while your current directory is in a PFS file system, a core file of length 0 is created.
- PFS does not support the **quotaon** or **sysacct** commands or the **mmap()** system call.
- PFS file regions cannot be locked by the **fcntl()** system call. However, you can use the **flock()** system call to lock the entire file.
- The maximum number of open files per process at any given time is 64. This includes the standard input, standard output, and standard error. This means that there is a practical maximum of 61 open files per process.

Using PFS Commands

In general, you use standard OSF/1 commands such as **ls**, **cat**, **cp**, and **mv** to manipulate files in PFS file systems. See the *OSF/1 Command Reference* for information on these commands. (Many commands do not work with files larger than 2G – 1 bytes, as described under "Using Extended Files" on page 5-33.) This section describes the additional file and file system commands provided by Paragon OSF/1.

Displaying File System Attributes

Command Synopsis	Description
showfs [**-k**] [**-t** *type*] [*filesystem* | *directory*]	Display file system attributes.

The command **showfs** with no arguments lists the file systems on your system, together with information on each. For example:

```
% showfs
Mounted on          512-blks     avail  capacity  sunit sfactor
/                    1458308    719276    45%
/home                4060838   3373782     8%
/usr                 2379194   1948124     9%
/home/.sdirs/vol0     598622    574464     4%
/home/.sdirs/vol1     598622    574464     4%
/home/.sdirs/vol2     598622    574464     4%
/home/.sdirs/vol3     598622    574464     4%
/pfs                 2394488   2297856     4%      8192       4
    sdirectories: /home/.sdirs/vol0
                  /home/.sdirs/vol1
                  /home/.sdirs/vol2
                  /home/.sdirs/vol3
```

In this case, the system has eight file systems. The seven file systems mounted on the directories */* (root), */home*, */usr*, */home/.sdirs/vol0*, */home/.sdirs/vol1*, */home/.sdirs/vol2*, and */home/.sdirs/vol3* are non-parallel file systems (type UFS or NFS); the file system mounted on the directory */pfs* is a PFS file system.

NOTE

There's nothing special about the name */pfs*; your PFS file systems can have any name. However, the rest of this chapter uses the convention that pathnames beginning with */pfs* are in a PFS file system.

The **showfs** command shows the following information for every file system:

`Mounted on`	The directory where the file system is mounted (its *mount point*). If you need to know the file system's device name, use the standard OSF/1 command **mount** or **df**.
`512-blks`	The total capacity of the file system in 512-byte disk blocks.
`avail`	The number of disk blocks currently available.
`capacity`	The approximate percentage of the file system's capacity currently in use.

In this example, the file system mounted on */usr* has a size of 2,379,194 512-byte disk blocks, of which 1,948,124 blocks are currently unused, so that the file system is approximately 9% full.

The **showfs** command shows the following additional information for each PFS file system:

`sunit`	The file system's stripe unit, in bytes.
`sfactor`	The number of stripe directories within the PFS file system.
`sdirectories`	The stripe directories (usually mount points of UFS file systems) within the PFS file system.

In this example, the PFS file system mounted on */pfs* has a stripe unit of 8K bytes and consists of the four UFS file systems mounted on */home/.sdirs/vol0*, */home/.sdirs/vol1*, */home/.sdirs/vol2*, and */home/.sdirs/vol3*.

The **showfs** command accepts the following optional arguments:

-k	Display capacity and available capacity in 1024-byte disk blocks instead of 512-byte disk blocks. The header "512-blks" changes to "kbytes".
-t *type*	Display information about all file systems of type *type*, where *type* is any recognized file system type in lowercase (**pfs**, **ufs**, or **nfs**).
filesystem	Display information about the file system whose device name is *filesystem*.
directory	Display information about the file system mounted on *directory*.

The *filesystem* or *directory* argument overrides **-t** *type* if used together.

NOTE

You should use **showfs**, not **df**, to get information about the cumulative amount of free space in a PFS file system. Using the standard **df** command on a PFS file system only gives information about the single disk partition on which the PFS file system is mounted, so does not indicate how much space is actually available for file striping.

Increasing the Size of a File

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

The **lsize** command changes the amount of disk space allocated to each specified file. You can use this command to allocate all the space you will need for a large file before you run the application that writes to the file. This makes sure that there is enough room in the file system for the file, and can also increase file I/O performance.

The **lsize** command has two forms:

lsize *size file* [*file* ...]	Sets the size of 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 (its contents are undefined).

lsize will not decrease the size of a file. If the specified size is smaller than the file's current size, the command has no effect.

Using Parallel I/O Calls

The rest of this chapter discusses the *parallel I/O calls* you can use in parallel applications to access both PFS and non-PFS files.

The term *parallel I/O calls* refers to all I/O calls that are provided by Paragon OSF/1 but not by standard OSF/1. These calls facilitate I/O on multiple nodes and permit I/O to very large files in PFS file systems. They are part of the library *libnx.a*, which is automatically searched when you link an application with the **-nx** switch. You can also use the switch **-lnx** to search *libnx.a* without using **-nx**. See "Compiling and Linking Applications" on page 2-5 for more information on these switches.

Most of the parallel I/O calls can only be used in programs running in the compute partition. They will not work, or will give unexpected results, if used in a program running in the service partition. (See "Managing Partitions" on page 2-25 for more information on the service and compute partitions.)

NOTE

The parameter *fileID* in the system call synopses in this chapter is an integer that represents an open file: a *unit* in Fortran, or a *file descriptor* in C.

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

Opening Files in Parallel

Synopsis		Description
gopen(*path*, *oflag*, *iomode* [, *perms*])	(C)	Open a file on all nodes and set its I/O mode.
gopen(*unit*, *path*, *iomode***)**	(Fortran)	

To open a file for use by all the nodes in your application, call **gopen()**. You can use **gopen()** to open files in both PFS and non-PFS file systems. **gopen()** works like the standard **open()** operation, with the following exceptions:

- It is a *global call*. All the nodes in the application must call **gopen()**, and all must call it with the same arguments.
- It is a *synchronizing call*. Each node blocks at the **gopen()** until all the nodes have called it.
- It sets the *I/O mode* of the file, as described under "Using I/O Modes" on page 5-13.
- When called on a large number of nodes, it offers better performance and causes less system overhead.

Note that **gopen**() must be called by *all* the nodes in the application, even those that do not actually perform any I/O. For example, suppose that your application has a "manager" node that assigns I/O work to the "worker" nodes, but does no I/O itself. If you want to use **gopen**(), all the nodes, even the manager, must open the file.

Using gopen() in C

The C version of **gopen()** opens the specified file and returns a file descriptor, like the standard OSF/1 system call **open()**. In addition to being a global synchronizing call and setting the I/O mode of the file, as discussed earlier, the C version of **gopen()** has the following differences from the standard **open()**:

- It can only be used to open an ordinary file (not a directory or a device special file).
- If an error occurs, it prints an error message and terminates the calling process.

gopen() is otherwise equivalent to **open()**. For example, the following C call opens the file */pfs/mydat* for reading and writing, creating it if it does not exist, and returns a file descriptor that you can use to access it. The file's I/O mode is set to **M_GLOBAL**, and if the file is created it is given permissions 644 octal (`rw-r--r--`).

```
#include <fcntl.h>
#include <nx.h>
int fd;
fd = gopen("/pfs/mydat", O_RDWR | O_CREAT, M_GLOBAL, 0644);
```

The symbolic names for *oflag* (such as **O_CREAT**) are defined in the header file *fcntl.h*, and the symbolic names for *iomode* (such as **M_GLOBAL**) are defined in the header file *nx.h*.

See **open()** in the *OSF/1 Programmer's Reference* for information on the *oflag* parameter; see "Using I/O Modes" on page 5-13 for information on the *iomode* parameter; see **chmod()** in the *OSF/1 Programmer's Reference* for information on the *perms* parameter.

Using gopen() in Fortran

The Fortran version of **gopen()** opens the specified file for unformatted I/O on a specified unit. It is equivalent to the following Fortran **open()** statement:

```
      OPEN(unit, path, status='unknown', form='unformatted',
     x     access='sequential')
```

However, it differs from the standard Fortran **open()** in that it is a subroutine. Also, as discussed earlier, it is a global synchronizing call and sets the I/O mode of the file.

For example, the following Fortran call opens the file */pfs/mydat* on unit 10 in I/O mode **M_GLOBAL**:

```
      include 'fnx.h'
      call gopen(10, "/pfs/mydat", M_GLOBAL)
```

The symbolic names for *iomode* (such as **M_GLOBAL**) are defined in the header file *fnx.h*.

Opening Files with Standard Operations

PFS and non-PFS files can also be opened and closed with the standard OSF/1 system calls and Fortran routines. For example, to open the file */pfs/mydat* for read and write access:

```
/* C version */
fd = open("/pfs/mydat", O_CREAT | O_RDWR, 0644);

c     Fortran version
      open(unit=10, file = '/pfs/mydat',
     x     status = 'new', form='unformatted')
```

Use this method when not all nodes open the same file at the same time, or when source compatibility with other systems is necessary. (Note that, if you want to use any synchronizing calls, all nodes must open the file.)

NOTE

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

The following section discusses additional special considerations for Fortran.

Special Considerations for Fortran

This section describes the special considerations that apply when you open files with the standard Fortran **open()** instead of **gopen()**.

Formatted Versus Unformatted I/O

If you call **open()** with **form='formatted'** (the default):

- You must use only Fortran I/O statements to access the file. You cannot use any of the parallel I/O calls described in this chapter on the file.
- Only one node may perform I/O to the file. If you perform formatted I/O to the same file from multiple nodes, the results are undefined.

If you open a file with **form='unformatted'**, you can use either Fortran I/O statements or parallel I/O calls to access the file. However, you must pick either one or the other: mixing Fortran I/O and parallel I/O to the same file can give unexpected results.

For the best I/O performance, you should use **gopen()**, or **open()** with **form='unformatted'**, and use parallel I/O calls for all file I/O.

If compatibility with other programs that use formatted I/O is required, you can perform formatted I/O to an internal file or a string and then use **cwrite()** to write the data to a file. However, if you use a string you must add a newline (ASCII character 10) to the end of the string using the function **char()**, since neither formatted I/O to a string nor **cwrite()** will add these for you. For example:

```
      include 'fnx.h'
      character*20 msgbuffer

      write(msgbuffer, 26) answer, char(10)
26    format('The answer is: ', i4, a1)
      call cwrite(10, msgbuffer, 20)
```

Alternatively, you can write a small program that translates your data files from unformatted to formatted and vice versa, and run it only when you need to share data with other programs.

New Files

If you call **open()** with **status='new'**, the result depends on whether or not the program is running on multiple nodes:

- If the program is running on one node (**numnodes()** is 1 or undefined), the **open()** fails if the file exists, as specified by the ANSI standard.

- If the program is running on multiple nodes (**numnodes()** is greater than 1), the file is truncated if it exists, as though you had specified **status='unknown'**.

This change makes it possible to specify **status='new'** when multiple nodes are opening a file that does not yet exist; with the standard Fortran semantics for **status='new'**, the first node to execute the **open()** statement would create the file, and the other nodes would fail because the file already exists. You can use the system call **stat()** to determine if a file exists before you open it.

Unnamed Files

If you call **open()** with no filename, the result depends on whether or not you specified **status='scratch'**:

- If you did not specify **status='scratch'**, the file is created in the current working directory with the filename *fort.nnn*, where *nnn* is the unit number. The file remains after the program terminates.

- If you specified **status='scratch'**, the file is created in the directory */usr/tmp* with the filename *FTNxxxxxxxx.nn*, where *xxxxxxxx* is the OSF/1 process ID of the creating process and *nn* is the unit number. The file does not remain after the program terminates, whether it terminated normally or abnormally.

For compatibility with the iPSC system, if you specified **status='scratch'** and the directory specified by the variable *CFS_MOUNT* exists (or, if *CFS_MOUNT* is not defined, if the directory */cfs* exists), the file *FTNxxxxxxxx.nn* is created in *$CFS_MOUNT* (or */cfs*) instead of */usr/tmp*.

Using I/O Modes

Synopsis	Description
setiomode(*fileID*, *iomode***)**	Set the I/O mode for a file.
iomode(*fileID***)**	Return the current I/O mode for a file.

A parallel application can access a file in one of five I/O modes. You can specify a file's I/O mode when you open it with **gopen()**, and you can use **setiomode()** to change the I/O mode of a file that is already open. You can use **iomode()** to determine an open file's current mode.

Like **gopen()**, **setiomode()** is a global synchronizing call. When a node calls **setiomode()**, it blocks until all the other nodes in the application call **setiomode()** with the same arguments. **setiomode()** must be called by *all* the nodes in the application, even those that do not actually perform any I/O (this means that all nodes must open the file). Also, **setiomode()** can only be used on an ordinary file, not a directory or a device special file.

A file's I/O mode actually belongs to the file descriptor or unit through which the file is accessed, not to the file itself. The I/O mode is not stored with the file, and different programs can access the same file with different I/O modes (even at the same time).

A file's I/O mode is *not* inherited across a **fork()** (after a **fork()** all files in the child process have I/O mode **M_UNIX**).

There are five I/O modes, each of which has a name and a number:

M_UNIX (0) In this mode, each node has its own file pointer and file operations are performed on a first-come, first-served basis. If you open a file with the C **open()** call or Fortran **open** statement, it is opened with this mode (but you can change it with **setiomode**()).

M_LOG (1) In this mode, all nodes share the same file pointer and file operations are performed on a first-come, first-served basis.

M_SYNC (2) In this mode, all nodes share the same file pointer and file operations are performed in order by node number. Records may be of variable length.

M_RECORD (3)
In this mode, each node has its own file pointer and file operations are performed on a first-come, first-served basis. However, records are stored in the file in order by node number. Records must be of a fixed length.

M_GLOBAL (4)
In this mode, all nodes share the same file pointer and must perform the same file operations at the same time. All file operations are performed by a single node, which then distributes the results to the other nodes over the internode network.

The names **M_UNIX**, **M_LOG**, **M_SYNC**, **M_RECORD**, and **M_GLOBAL** 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 5-48.

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. This mode is the default for files opened with the standard **open** call or statement.

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, the operating system on each node can determine where in the file it should be reading from or writing to independently of the other nodes. The results of reading or writing a file with **M_RECORD** are the same as **M_SYNC**, but **M_RECORD** is more efficient because no synchronization is required. No seeking is required by the application; the file system automatically reads or writes file data to or from the proper offset in the file.

  For example, suppose node 2 in an application running on four nodes writes a 10-byte record. Node 2's file pointer is first moved forward by 20 bytes to leave room for the records from nodes 0 and 1. Next, node 2's record is written to the file (which advances the file pointer by 10 bytes). Finally, node 2's file pointer is moved forward by 10 bytes to leave room for node 3's record. The other nodes can fill in their "slots" at any time (earlier or later); no synchronization or communication between nodes is required.

- Closing a file is a synchronizing operation, as for **M_LOG** and **M_SYNC**.
- As for **M_SYNC**, **lseek()** becomes a synchronizing call, and the only valid use for **lseek()** is for all nodes to seek to the same position in the file. If nodes attempt to seek to different positions, an error occurs.

M_GLOBAL (Mode 4)

In mode **M_GLOBAL** (number 4), all nodes must read and write the same data to the same parts of the file at the same time. This mode gives excellent performance for programs that work this way, such as a program where every node reads in the entire contents of a large input file.

- All nodes share a single file pointer.
- All the nodes in the application must open the file, and all must perform the same operations on the file at the same time.
- All file operations are synchronizing.
- Corresponding reads and writes must be of the same size on all nodes.
- The only valid use for **lseek()** is for all nodes to seek to the same position in the file.
- When the nodes write to a file, only the data written by a single node is actually written. Data written by other nodes is ignored.

The way that this mode is implemented is that only a single node actually reads from and writes to the disk. After a read, that node distributes the data to the other nodes over the internode network. This eliminates the contention for the disk device that would otherwise occur when many nodes attempt to read from the same place in a file at the same time.

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), and sends it to the other nodes. Then all nodes call **gopen()** to open the file *mydat* in the current directory (which could be in either a PFS file system or a non-PFS file system) with the specified I/O mode.

Each node then writes 10 records to the file. Each record contains the time in seconds since the file was opened, to four decimal places, and the message "Hello from node *x*." Node 0 waits one second before each write to the file; the other nodes write as fast as they can (this demonstrates how writes to the file are differently synchronized in the different modes). When each node finishes writing, it writes a "done writing" message to the screen. Then it closes the file and writes a "finished" message to the screen (the two messages show that, in some modes, **close()** is a synchronizing operation).

Fortran Example

```
      program iomodes

      include 'fnx.h'

      integer nunit, mode, iam
      double precision start, now, loop_time, loop_start
      character*16 msg
      character*29 msgbuffer

      msg = 'Hello from node '
      nunit = 12
      iam = mynode()

      if(iam .eq. 0) then
         print *, 'Enter I/O mode (0, 1, 2, 3, or 4):'
         read(*, 11) mode
11       format(i1)
         call csend(1, mode, 4, -1, myptype())
      else
         call crecv(1, mode, 4)
      endif

      call gopen(nunit, "mydat", mode)
      print 13, iam, iomode(nunit)
13    format('Node ', i4, ' using mode ', i1)

      start = dclock()
      do 100 i = 1, 10
c        *** if node 0, do nothing for 1.0 seconds ***
         if(iam .eq. 0) then
            loop_start = dclock()
101         loop_time = dclock() - loop_start
            if (loop_time .lt. 1.0) goto 101
         endif
```

```
c        *** all nodes now write a record to the file ***
102      now = dclock() - start
         write(msgbuffer, 14) now, msg, iam, char(10)
14       format(f7.4, a17, i4, a1)
         call cwrite(nunit, msgbuffer, 29)
100   continue

      print 15, iam
15    format('Node ', i3, ' done writing')
      close(nunit)
      print 16, iam
16    format('Node ', i3, ' finished')
      end
```

C Example

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

main()
{
    int    i, fd;
    double start, now;
    double loop_start, loop_cur;
    long   mode, iam;
    char   instring[40], msg[40];

    iam = mynode();

    if(iam == 0) {
        printf("Enter I/O mode (0, 1, 2, 3, or 4):\n");
        gets(instring);
        sscanf(instring, "%ld", &mode);
        csend(1, &mode, sizeof(mode), -1, myptype());
    } else {
        crecv(1, &mode, sizeof(mode));
    }

    fd = gopen("mydat", O_WRONLY | O_CREAT | O_TRUNC, mode, 0666);
    printf("Node %d using mode %d\n", iam, iomode(fd));
```

```
    start = dclock();
    for(i=0;i<10;i++) {
        if(iam==0) {
            loop_start = dclock();
            loop_cur = loop_start;
            while(loop_cur - loop_start < 1.0) {
                loop_cur = dclock();
            }
        }
        now = dclock() - start;
        sprintf(msg, "%7.4f Hello from node %4ld\n", now, iam);
        cwrite(fd, msg, strlen(msg));
    }

    printf("Node %d done writing\n", iam);
    close(fd);
    printf("Node %d finished\n", iam);
}
```

Compiling and Running the Example

To compile this program to a parallel application, use the following **if77** or **icc** command:

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

or

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

When you run the resulting application, you may find the output easier to understand if you run the example on four or fewer nodes. Use the **-sz** switch to determine the number of nodes on which the application runs (see “Controlling the Application’s Execution Characteristics” on page 2-13 for information on **-sz** and other application switches).

For example, to run the application on two nodes of your default partition with I/O mode 1 (**M_LOG**):

```
% iomodes -sz 2
Enter I/O mode (0, 1, 2, 3, or 4):
1
Node 0 using mode 1
Node 1 using mode 1
Node 1 done writing
Node 0 done writing
Node 1 finished
Node 0 finished
%
```

The following example outputs came from the C version of the example, run on two nodes.

M_UNIX Output

In mode **M_UNIX** (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.

M_GLOBAL Output

In mode **M_GLOBAL** (4), writes by all nodes but one (node 0 in this case) are ignored. As a result, the file contains only the writes from that node.

```
 1.0000 Hello from node    0
 2.0087 Hello from node    0
          •
          •
          •
 9.0711 Hello from node    0
10.0797 Hello from node    0
```

This output is the same as the output of **M_UNIX**, but the other nodes do not compete with node 0 for access to the disk, so this mode is more efficient. However, because this program uses such a small data file, the difference in execution time is probably not noticeable.

Note that **M_GLOBAL** is usually used for reading, not writing.

Reading and Writing Files in Parallel

You can read and write files with the familiar OSF/1 system calls and Fortran routines. For example, here is a Fortran code fragment that opens a file whose pathname is */pfs/mydat* and reads some data into an array called *array* using the Fortran **read** statement:

```
open(unit=10, file='/pfs/mydat', form='unformatted')
read 10, (array(j), j=1, n)
```

In addition to the usual I/O facilities, the Paragon OSF/1 operating system offers a series of parallel I/O calls, which are discussed in the following pages. These calls can be used on files in both PFS and non-PFS file systems.

Like the message-passing calls, the parallel I/O calls offer you the choice of *synchronous* or *asynchronous* I/O. The synchronous calls begin with **c** (for "complete") and do not return until the operation is complete. The asynchronous calls begin with **i** (for "incomplete") and return immediately; you use the call **iodone()** or **iowait()** to determine when the operation is complete.

If you program in Fortran, you should use the parallel I/O calls rather than Fortran I/O whenever you can. These calls offer better performance than the Fortran I/O routines, and you can test for the end of a file with **iseof()**. (This does not apply to C programmers; the usual C I/O calls are as efficient as their parallel I/O counterparts.) However, if you use parallel I/O calls on a file, you must not use Fortran file I/O statements on the same file (for example, you must not mix **write** and **cwrite()** on the same file).

NOTE

Parallel I/O to NFS files may give poor performance or unexpected results.

The 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.
creadv(*fileID*, *iov*, *iovcnt***)**	Read from a file to irregularly-scattered buffers, waiting for completion.
cwritev(*fileID*, *iov*, *iovcnt***)**	Write to a file from irregularly-scattered buffers, waiting for completion.

The calls **cread()**, **cwrite()**, **creadv()**, and **cwritev()** perform synchronous file I/O. They are equivalent to the standard OSF/1 calls **read()**, **write()**, **readv()**, and **writev()**, except that they follow the same naming and error-handling conventions as the 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 4-42 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 */pfs/hello*:

```
fd = open("/pfs/hello", O_RDWR, 0644);
         •
         •
         •
sprintf(buffer, "Hello from node %d\n", iam);
cwrite(fd, buffer, strlen(buffer));
```

Here is a slightly more complicated example: a Fortran code fragment that opens a file whose pathname is */pfs/mydat*, seeks to a location, and reads some data using the synchronous call **cread()**. The data represents a matrix stored in rows of *n* four-byte elements. Each node reads *m* rows and performs a calculation with each row (calling the Basic Linear Algebra Subroutines routine **sdot()** to get the dot product of two vectors). Because each node seeks to a different place in the file, you must use I/O mode **M_UNIX** (the default).

```
      open(unit=10, file='/pfs/mydat', form='unformatted')
      lseek(10, 4*mynode()*n*m, 0)

      do 10 i = 1, m
         call cread(10, arow, n*4)
         y(i) = sdot(n, arow, 1, xtotal, 1)
10    continue
```

Note that when you open a file in Fortran, you must open it as sequential and unformatted to be able to use **cread()** and **cwrite()**. (Sequential is the default access, but you must specify **form='unformatted'**.)

NOTE

Unlike their OSF/1 equivalents, these calls do not return the number of bytes read or written. If any error occurs, these calls print an error message and terminate the calling process.

Reading past the end of a file is considered an error, so you must be certain you know how many bytes remain in the file before you read from it. You can use **iseof()**, to detect end-of-file, after each **cread()** or **creadv()**. You can also use the following call to determine the length of a file:

```
length = lseek(unit, 0, SEEK_END)
```

This call sets the file pointer to the end of the file and returns the current position of the file pointer (that is, the file's length). You can then use **lseek(unit, 0, SEEK_SET)** to return the file pointer to the beginning of the file. (If the file might be larger than 2G – 1 bytes, use **eseek()** instead of **lseek()**; see "Manipulating Extended Files" on page 5-36 for more information.)

If you need to detect errors in reading and writing, you must program in C and use either the standard OSF/1 calls (**read()**, **write()**, **readv()**, and **writev()**, described in the *OSF/1 Programmer's Reference*) or the underscore versions of the parallel I/O calls (**_cread()**, **_cwrite()**, **_creadv()**, and **_cwritev()**, described under "Handling Errors" on page 4-42). The underscore versions do return the number of bytes read or written.

Asynchronous File I/O

Synopsis	Description
iread(*fileID*, *buffer*, *nbytes***)**	Read from a file without waiting for completion.
iwrite(*fileID*, *buffer*, *nbytes***)**	Write to a file without waiting for completion.
ireadv(*fileID*, *iov*, *iovcnt***)**	Read from a file to irregularly-scattered buffers, without waiting for completion.
iwritev(*fileID*, *iov*, *iovcnt***)**	Write to a file from irregularly-scattered buffers, without waiting for completion.
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()**, **iwrite()**, **ireadv()**, and **iwritev()** perform asynchronous file I/O. They work like **cread()**, **cwrite()**, **creadv()**, and **cwritev()**, but they return immediately, without waiting for the read or write to complete. The asynchronous I/O calls return an I/O ID much like the message ID returned by the asynchronous message passing calls. You can pass this I/O ID to **iodone()** or **iowait()** to determine when the asynchronous file I/O operation has completed.

NOTE

The number of I/O IDs is limited, so you *must* use **iodone()** or **iowait()** to release each ID after you use it.

To check if an asynchronous I/O operation has completed, use the **iodone()** call. It returns 1 if the asynchronous operation has completed and 0 otherwise. You can also decide to block on the completion of an asynchronous call. Use the **iowait()** call for this. Both **iodone()** and **iowait()** take the I/O ID as an input parameter. For example (in Fortran):

```
c    Write to a file
         ioid = iwrite(12, sbuf, size)
         •
         •
         •
c    Do some calculation...
         •
         •
         •
c    Wait until the write completes
         call iowait(ioid)
```

The number of available I/O IDs is limited; be sure to release IDs that are no longer needed. There are two ways to release an I/O ID: you can issue an **iowait()**, as shown in the previous example, or you can keep issuing **iodone()**s until an **iodone()** returns 1.

NOTE

To preserve data integrity, all I/O requests that use or affect the file pointer are processed on a "first-in, first-out" basis.

This means that if an asynchronous I/O call is followed by a synchronous read, write, or seek on the same file, the synchronous call will block until the asynchronous operation has completed.

Closing Files in Parallel

It's always a good idea to close a file when you are finished using it. Whether you used **open()** or **gopen()** to open a file, and whether the file is a PFS file or a non-PFS file, you use the standard OSF/1 system calls or Fortran routines to close it.

For example, to close the file open on file descriptor *fd* (C) or unit 10 (Fortran):

```
/* C version */
close(fd);

c     Fortran version
      close(unit=10)
```

NOTE

If the I/O mode of the file being closed is anything other than **M_UNIX**, closing the file is a synchronizing operation.

See "Using I/O Modes" on page 5-13 for more information.

Detecting End-of-File and Moving the File Pointer

Synopsis	Description
iseof(*fileID***)**	Test for end-of-file.
lseek(*fileID*, *offset*, *whence***)**	Move the read/write file pointer.

The calls **iseof()** and **lseek()** are provided for both C and Fortran programmers. If you use parallel I/O calls to perform file I/O in a Fortran program, you must use **iseof()** and **lseek()** instead of the equivalent Fortran features.

The **iseof()** call returns 1 if the given file is at the end of the file and 0 otherwise. For example, the following Fortran code reads characters from the file open on unit 12, writing each one to the screen, until it reaches the end of the file:

```
      do while(iseof(12) .eq. 0)
         call cread(12, char, 1)
         print 300, iam, char
300      format('Node ', i3,' read:  ', a1)
      end do
```

The **lseek()** call moves the file pointer to *offset* bytes from the point specified by *whence*, which can be either a name or a number:

- If *whence* is **SEEK_SET**, **lseek()** moves the pointer to *offset* bytes from the beginning of the file.
- If *whence* is **SEEK_CUR**, **lseek()** moves the pointer forward *offset* bytes from its current position.
- If *whence* is **SEEK_END**, **lseek()** moves the pointer to *offset* bytes after the end of the file.

The names **SEEK_SET**, **SEEK_CUR**, and **SEEK_END** are constants defined in the header files *unistd.h* (for C) and *fnx.h* (for Fortran). For compatibility with the iPSC system, the numeric values 0, 1, and 2 are also accepted (but using the symbolic names is recommended).

lseek() returns the new position of the file pointer (measured in bytes from the beginning of the file).

For example, the following C call moves the file pointer of the file open on file descriptor *fd* to the beginning of the file:

```
#include <unistd.h>

newpos = lseek(fd, 0, SEEK_SET);
```

The following Fortran call moves the file pointer of the file open on unit 12 forward 500 bytes:

```
        include 'fnx.h'

        newpos = lseek(12, 500, SEEK_CUR)
```

NOTE

If the I/O mode of the file is **M_SYNC**, **M_RECORD**, or **M_GLOBAL**, seeking is a synchronizing operation.

See "Using I/O Modes" on page 5-13 for more information.

Flushing Fortran Buffered I/O

Synopsis	Description
forceflush()	Cause all buffered I/O to be flushed if an exception occurs.
forflush(*unit***)**	Flush all buffered I/O on a particular unit.

The subroutines **forceflush()** and **forflush()** let Fortran programmers make sure that buffered I/O actually goes to the associated file or device. These subroutines are not available to C programs.

Fortran I/O to files and devices other than the user's terminal is *buffered*—that is, when you write to a file, the data is stored in a memory buffer, and only written to the corresponding file or device when the buffer is full. However, if another node is waiting for some data to appear in a file, you might want to force the contents of a unit's buffer to be written immediately. You can do this by calling **forflush()** on the unit. For example, to flush all buffered I/O on unit 9 to the corresponding file or device:

```
call forflush(9)
```

Another possible problem with buffered I/O is that if the program is interrupted by an exception, buffered data that has not yet been written to the file is lost. The subroutine **forceflush()** establishes a signal handler that flushes all buffered I/O in case of an exception. You call it as follows:

```
call forceflush
```

Note that you must call **forceflush()** *before* the exception occurs. You can use **fpsetmask()** (described under "Controlling Floating-Point Behavior" on page 4-46) to control whether or not an exception occurs in case of certain floating-point errors.

Fortran I/O to the user's terminal is not buffered. You can avoid buffering to files and devices by using parallel file I/O calls such as **cwrite()** and **iwrite()** instead of Fortran I/O. These calls do not buffer I/O into the Fortran I/O memory buffer; when the call returns, you can be sure the data has been sent to the specified file or device. (However, there may be some buffering within the operating system, which cannot be avoided.)

Using "###" Filenames

If you perform certain standard file operations (shown in Table 5-1) on a file that contains three or more # symbols in its filename, the series of # symbols is automatically replaced by the node number (within the application) of the node that opens the file.

Table 5-1. File Operations that Accept "###" Filenames

access()	**mknod()**	**truncate()**
chdir()	**open()**	**unlink()**
chmod()	**readlink()**	**utimes()**
chown()	**rmdir()**	**link()**
creat()	**stat()**	**rename()**
mkdir()	**statfs()**	**symlink()**

The Fortran equivalents of these operations also support "###" filenames. Note, though, that **gopen()** does *not* appear in this list.

For example, assume that you have the same program running on all the nodes of your application, and each node calls **open()** to open a file called *file###*. The result is that each node opens a separate file. Node 0 opens *file000*, node 1 opens *file001*, node 2 opens *file002*, and so on. If an application opens *file###* for reading, the specified files (*file000*, *file001*, *file002*, and so on) must exist.

If you use a "###" filename in a non-parallel program running in the service partition, it uses node number 0. For example, opening a file called *file###* from a service node opens *file000*. Note that this also affects standard commands that make these calls; for example, since the **rm** command calls **unlink**(), the command **rm file###** will attempt to remove the file *file000*.

Filenames containing a sequence of one or two **#** symbols are not affected. For example, the file *file##* is a single file that is accessible by each node.

If the number of digits in a node number is less than the number of # symbols in the filename, the node number is padded with zeros to the length of the sequence of # symbols. If the number of digits in a node number exceeds the number of # symbols in the filename, the filename is extended, but only when necessary. For example, calling **unlink()** on the file *data.###* in every node of an application running on 2000 nodes unlinks files *data.000*, *data.001*, *data.002 ... data.998*, *data.999*, *data.1000*, *data.1001 ... data.1998*, and *data.1999*.

There is nothing special about files created in this way; each file created is a single ordinary file. For example, suppose an application uses **open**() or **creat**() to create *###myfile*, writes into it, and then closes the file. This creates a series of files called *000myfile*, *001myfile*, *002myfile*, and so on. Each of these files is an ordinary file; for example, you can delete one without affecting the others, and there's nothing to prevent node 1 from opening *005myfile*.

Increasing the Size of a File

Synopsis	Description
lsize(*fileID*, *offset*, *whence***)**	Increase size of a file.

You can allocate more space to a file with **lsize()**. The **lsize**() call sets the file's size as specified by *offset* and *whence*:

- If *whence* is **SIZE_SET**, **lsize()** sets the file's size to *offset* bytes.
- If *whence* is **SIZE_CUR**, **lsize()** sets the file's size to the current file pointer position plus *offset* bytes.
- If *whence* is **SIZE_END**, **lsize()** increases the file's size by *offset* bytes.

The names **SIZE_SET**, **SIZE_CUR**, and **SIZE_END** are constants defined in the header files *nx.h* (for C) and *fnx.h* (for Fortran). For compatibility with the iPSC system, the numeric values 0, 1, and 2 are also accepted (but using the symbolic names is recommended).

For example, the following Fortran call increases the size of the file open on *unit1* to one million bytes:

```
include 'fnx.h'

size = lsize(unit1, 1000000, SIZE_SET)
```

The following C call increases the size of the file open on file descriptor *fd* by 500,000 bytes:

```
#include <unistd.h>
#include <nx.h>
int size, fd;

size = lsize(fd, 500000, SIZE_CUR)
```

The additional space is allocated to the file from the file system, but it is not initialized (its contents are undefined).

lsize() will not decrease the size of a file. If the size specified by *offset* and *whence* is smaller than the file's current size, the call has no effect.

The major use of this call is to ensure that enough disk space is available before you begin a lengthy calculation. Pre-allocating disk space can also improve disk performance.

Using Extended Files

A PFS file greater than or equal to 2G bytes in size is called an *extended file*. These files are stored in the same way as non-extended PFS files. However, some of the file parameters (like the file pointer and file size) do not fit into a 32-bit integer. This means that standard OSF/1 calls and commands that use these parameters cannot be used on extended files. The following two sections list the calls and commands that do not support extended files.

OSF/1 Calls that Do Not Support Extended Files

Most OSF/1 calls, such as **read**() and **write()**, don't care how big the file is and work perfectly well on extended files. The OSF/1 calls that have problems with extended files are shown in Table 5-2.

Table 5-2. OSF/1 Calls Not Supporting Extended Files

Call	Problem
fcntl()	Can't lock a file region larger than 2G – 1 bytes.
fgetpos()	Can't return an offset greater than 2G – 1 bytes.
fseek()	Can't specify an offset greater than 2G – 1 bytes.
unlocked_fseek()	Can't specify an offset greater than 2G – 1 bytes.
fsetpos()	Can't return an offset greater than 2G – 1 bytes.
fstat()	Can't be used on a file larger than 2G – 1 bytes.[1]
ftell()	Can't return an offset greater than 2G – 1 bytes.
ftruncate()	Can't specify a file size greater than 2G – 1 bytes.
lseek()	Can't specify an offset greater than 2G – 1 bytes.
lstat()	Can't be used on a file larger than 2G – 1 bytes.[1]
madvise()	Can't map a file larger than 2G – 1 bytes.
mmap()	Can't map a file larger than 2G – 1 bytes.
mprotect()	Can't map a file larger than 2G – 1 bytes.
msync()	Can't map a file larger than 2G – 1 bytes.
munmap()	Can't map a file larger than 2G – 1 bytes.
stat()	Can't be used on a file larger than 2G – 1 bytes.[1]
truncate()	Can't specify a file size greater than 2G – 1 bytes.

1. If you call **fstat**(), **lstat**(), or **stat()** on a file larger than 2G – 1 bytes, the call fails with the error **EFBIG**.

To manipulate extended files, Paragon OSF/1 provides special calls that perform the equivalent of **lseek**(), **stat**(), **fstat**(), and **lsize()** for extended files. These are discussed under "Manipulating Extended Files" on page 5-36.

OSF/1 Commands that Do Not Support Extended Files

Many OSF/1 commands make one or more of the system calls in Table 5-2, so do not work on extended files. The commands **chgrp**, **chmod**, **chown**, **cp**, **ls**, **mv**, **tar**, and **rm** have been specially modified to support extended files; most other commands will fail if used on extended files. (Note that you must use the **-E** switch to archive an extended file with **tar**; see **tar** in the *Paragon™ Commands Reference Manual* for more information.)

Table 5-3 shows the OSF/1 commands that are known to have problems with extended files. (This list is not guaranteed to be complete; other commands, not listed here, may also have problems.)

Table 5-3. OSF/1 Commands Not Supporting Extended Files

Command	Problem
cat	Can't read a file larger than 2G – 1 bytes.
compress	Can't compress a file larger than 2G – 1 bytes.
cpio	Can't handle files or archives larger than 2G – 1 bytes.
diff	Can't compare a file larger than 2G – 1 bytes.
du	Can't show the size of a directory containing files larger than 2G – 1 bytes.
ed	Can't edit a file larger than 2G – 1 bytes.
ex	Can't edit a file larger than 2G – 1 bytes.
find	Can't find a file larger than 2G – 1 bytes with **-size**; can't show the size of a file larger than 2G – 1 bytes with **-ls**.
ftp	Can't copy a file larger than 2G – 1 bytes.
more	Can't display a file larger than 2G – 1 bytes.
newfs	Can't create a UFS file system larger than 2G – 1 bytes.
rcp[1]	Can't copy a file larger than 2G – 1 bytes.
tail	Can't display a file larger than 2G – 1 bytes.
vi	Can't edit a file larger than 2G – 1 bytes.

1. Note that although **rcp** cannot copy an extended file, **cp** can.

Manipulating Extended Files

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

The **e...()** calls perform file operations on extended files. They do this by having parameters that are *extended integers* (a data type capable of representing integers greater than 2G – 1). You must use the calls described under "Performing Extended Arithmetic" on page 5-37 to operate on extended integers.

- The call **eseek()** is like **lseek()** (discussed under "Detecting End-of-File and Moving the File Pointer" on page 5-29), except that the *offset* parameter is an extended integer. The C version of this call is a function that returns the new position as an extended integer; the Fortran version is a subroutine that stores the new position in its fourth parameter.

- The call **esize()** is like **lsize()** (discussed under "Increasing the Size of a File" on page 5-32), except that the *offset* parameter is an extended integer. The C version of this call is a function that returns the new size as an extended integer; the Fortran version is a subroutine that stores the new size in its fourth parameter.

- The calls **estat()**, **lestat()**, and **festat()** are like the standard OSF/1 calls **stat()**, **lstat()**, and **fstat()** (described in the *OSF/1 Programmer's Reference*), except that they use a structure called *estat*, defined in *<sys/estat.h>*, which is the same as the OSF/1 *stat* structure except that the file size is an extended integer. These calls are available only in C, not in Fortran.

You *must* use these calls to manipulate extended files (PFS files greater than or equal to 2G bytes in size). However, you can also use these calls on non-PFS files and on PFS files less than 2G bytes in size. You can use these calls or the standard OSF/1 calls on PFS files less than 2G bytes in size.

Performing Extended Arithmetic

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

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

Extended integers are signed 64-bit integers with values from ($2^{63} - 1$) to -2^{63} (2^{63} is approximately 9.2×10^{18}).

- In Fortran, extended integers are stored in a two-element array of type **integer*4**.
- In C, extended integers are stored in a variable of type *esize_t*, a structure type defined in the header file *<sys/estat.h>*. (For compatibility with the iPSC system, there is also a header file *<estat.h>* that simply includes *<sys/estat.h>*.

You should always use extended arithmetic calls to operate on an extended integer, rather than access its internal structure.

Some of these calls return extended integers. The C versions of these calls return a value of type *esize_t*. However, Fortran does not allow functions to return arrays, so the Fortran versions of these calls are subroutines with an additional parameter: the result of the operation on the first two parameters is stored into the third parameter. For example, the following call adds the extended integers *e1* and *e2* and stores the result in *e_sum*:

```
/* C version */
#include <sys/estat.h>
esize_t e1, e2, e_sum;
e_sum = eadd(e1, e2);


c     Fortran version
      integer e1(2), e2(2), e_sum(2)
      call eadd(e1, e2, e_sum);
```

If you want to add an ordinary integer to an extended integer, you must create your own extended integer from the desired integer value. To create an extended integer, use **stoe()**. This call takes a string whose value is a number, and returns the corresponding numeric value as an extended integer. For example, the following code fragment adds 1 to the value of the extended integer *e1*. It does this by converting the string "1" to an extended integer with **stoe()**, storing the resulting extended integer in *e2*, and then adding *e2* to *e1* (note that in Fortran the string must be declared to be one character larger than the actual string being converted):

```
/* C version */
#include <sys/estat.h>
esize_t e1, e2, e_sum;
char *one = "1";

e2 = stoe(one);
e_sum = eadd(e1, e2);


c     Fortran version
      character*2 one
      parameter (one ='1')
      integer e1(2), e2(2), e_sum(2)

      call stoe(one, e2)
      call eadd(e1, e2, e_sum)
```

The other extended arithmetic calls allow you to subtract, multiply, divide, and find the remainder after division of extended integers. When you use **ediv()** or **emod()**, the divisor and answer must be 4-byte integers, not extended integers. Similarly, when you use **emul()**, the second argument must be a 4-byte integer, not an extended integer.

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

Getting Information About PFS File Systems

Synopsis	Description
getpfsinfo(*buf*)	Get PFS-specific information about all mounted PFS file systems.
statpfs(*path*, *fs_buffer*, *pfs_buffer*, *pfs_bufsize*)	Get PFS-specific and non-PFS-specific information for the file system containing *path*.
fstatpfs(*fildes*, *fs_buffer*, *pfs_buffer*, *pfs_bufsize*)	Get PFS-specific and non-PFS-specific information for the file system containing the file open on *fildes*.

The functions **getpfsinfo()**, **statpfs()**, and **fstatpfs()** let C programmers get information about PFS file systems. These functions are not available to Fortran programs. See "PFS File Systems and PFS Files" on page 5-3 for more information on the concepts discussed in this section.

Getting Information About All Mounted PFS File Systems

getpfsinfo() gets information about all mounted PFS file systems. It is similar to the standard OSF/1 call **getmntinfo()**, except that instead of returning information in an array of *statfs* structures, it returns information in an array of *pfsmntinfo* structures. It allocates the memory for this array of structures, each of which describes one PFS file system, and stores a pointer to this array into its argument. **getpfsinfo()** returns the number of elements in this array. The *pfsmntinfo* structure, defined in the header file *<pfs/pfs.h>*, contains the following fields:

m_mntonname Directory on which the PFS file system is mounted.

m_statpfs *statpfs* structure that describes the PFS file system.

The *statpfs* structure, also defined in the header file *<pfs/pfs.h>*, describes the PFS-specific attributes of a file system. This is a variable-size structure. It contains the following fields:

p_reclen Total size of this *statpfs* structure, in bytes.

p_sunitsize Stripe unit size for this PFS file system, in bytes.

p_sfactor Number of stripe directories within this PFS file system.

p_sdirs List of stripe directories within this PFS file system. The number of pathnames in the list is specified by *p_sfactor*.

Each pathname in *p_sdirs* is a structure of type *pathname_t* (defined in *<pfs/pfs.h>*); you can use the **NEXTPATH()** macro defined in *<pfs/pfs.h>* to examine each pathname in turn.

Here's an example of **getpfsinfo()**:

```
#include <sys/types.h>
#include <nx.h>
#include <pfs/pfs.h>

main() {
    struct pfsmntinfo *pfsinfo;
    struct statpfs    *sattr;
    pathname_t        *sdir;
    int                cnt, i, incr;

    cnt = getpfsinfo(&pfsinfo);

    if(cnt == 0) {
        printf("No PFS file systems mounted\n");
    } else {
        for(i = 0; i < cnt; i++) {
            printf("Mount point: %s\n", pfsinfo->m_mntonname);

            sattr = &(pfsinfo->m_statpfs);
            printf("  Stripe unit size: %d\n",
                   sattr->p_sunitsize);
            printf("  Stripe factor: %d\n", sattr->p_sfactor);

            sdir = &(sattr->p_sdirs);
            printf("  Stripe directories:\n");
            for(i = 0; i < sattr->p_sfactor; i++) {
                printf("    %s\n", sdir->name);
                sdir = NEXTPATH(sdir);
            }

            incr = sizeof(pfsinfo->m_mntonname)
                   + sattr->p_reclen;
            pfsinfo = (struct pfsmntinfo *)((char *)pfsinfo
                      + incr);
        }
    }
}
```

This program prints out the attributes of all mounted PFS file systems, something like the command **showfs -t pfs**. Note that you must use the **NEXTPATH()** macro to step through the *p_sdirs* field of the *statpfs* structure, and you must increment the pointer into the array of *pfsmntinfo* structures by the size of the current *pfsmntinfo* structure (using the value of its *p_reclen* field).

Getting PFS Information About a Single File System

statpfs() gets information about a file system given the pathname of a file or directory in that file system; **fstatpfs()** gets information about a file system given the file descriptor of an open file in that file system.

These functions get both general and PFS-specific information about the specified file system. They can be used on both PFS and non-PFS file systems, but they return PFS-specific information only for PFS file systems. They are similar to the standard OSF/1 calls **statfs()** and **fstatfs()**, except that instead of returning information in a *statfs* structure, they return information in an *estatfs* structure and a *statpfs* structure.

- The *estatfs* structure, defined in the header file *<pfs/pfs.h>*, describes the basic attributes of the file system. It is just like the *statfs* structure defined in *<sys/mount.h>*, except that some of its fields are of type *esize_t* (see "Performing Extended Arithmetic" on page 5-37 for information on this type). This is necessary because some of the values returned for PFS file systems are too large to be stored into an ordinary integer.

  Some of the more generally useful fields of the *estatfs* structure are:

f_type	The type of the file system, expressed as a constant such as **MOUNT_UFS**, **MOUNT_NFS**, or **MOUNT_PFS** (these constants are defined in *<sys/mount.h>*).
f_bavail	Number of free 1024-byte disk blocks in the file system available to ordinary users, expressed as a value of type *esize_t*.
f_mntonname	Directory on which the file system is mounted, expressed as a string.
f_mntfromname	Device name of the file system, expressed as a string.

  See **statpfs()** in the *Paragon™ C System Calls Reference Manual* for a complete description of all fields in the *estatfs* structure.

- The *statpfs* structure is the same *statpfs* structure described for **getpfsinfo()** in the previous section. However, the way it is returned is different: **getpfsinfo()** allocates space for several *statpfs* structures and returns you a pointer to this space, but **statpfs()** and **fstatpfs()** store information in a *statpfs* structure that you provide.

  Because the *statpfs* structure is variable-size, you must tell **statpfs()** and **fstatpfs()** how big your *statpfs* structure is; you do this with the third parameter of **statpfs()** and **fstatpfs()** (called *pfs_bufsize*). Then you must check the *p_reclen* field in the returned *statpfs* structure to be sure the returned information fit in your provided structure; if it didn't, try again with a larger structure.

Here's an example of **statpfs()**:

```
#include <sys/types.h>
#include <sys/mount.h>
#include <malloc.h>
#include <nx.h>
#include <pfs/pfs.h>

#define SDIRS_INIT_SIZE  1024

main(int argc, char **argv) {
    struct statpfs *statpfsbuf;
    int             bufsize;
    struct estatfs  estatbuf;
    pathname_t     *sdir;
    char            blocks[80];
    int             i;

    if(argc != 2)
    {
        printf("Usage: %s <mountpoint>\n", argv[0]);
        exit(1);
    }

    bufsize=sizeof(struct statpfs) + SDIRS_INIT_SIZE;

    statpfsbuf=(struct statpfs *)malloc(bufsize);

    if(statpfs(argv[1], &estatbuf, statpfsbuf, bufsize) < 0)
    {
        nx_perror("statpfs");
        exit(1);
    }

    if(statpfsbuf->p_reclen > bufsize)
    {
        bufsize=statpfsbuf->p_reclen;
        statpfsbuf=(struct statpfs *)realloc(statpfsbuf,
                                             bufsize);

        if(statpfs(argv[1], &estatbuf, statpfsbuf, bufsize)
           < 0)
        {
            nx_perror("statpfs");
            exit(1);
        }
    }
```

```
    printf("Selected PFS statistics for %s:\n", argv[1]);

    /* From estatfs structure */

    printf("  File system type: %d\n", estatbuf.f_type);
    etos(estatbuf.f_bavail, blocks);
    printf("  # of 1K blocks available: %s\n", blocks);
    printf("  Mount point: %s\n", estatbuf.f_mntonname);
    printf("  Device name: %s\n", estatbuf.f_mntfromname);

    /* From statpfs structure */

    printf("  Stripe unit size: %d\n",
           statpfsbuf->p_sunitsize);
    printf("  Stripe factor: %d\n", statpfsbuf->p_sfactor);

    printf("  Stripe directories:\n");
    sdir = &(statpfsbuf->p_sdirs);
    for (i = 0; i < statpfsbuf->p_sfactor; i++) {
        printf("    %s\n", sdir->name);
        sdir = NEXTPATH(sdir);
    }
}
```

This program prints out the attributes of the file system containing the file specified by its first argument. Note that you must allocate enough space for the *statpfs* structure plus the stripe directory pathnames and check the returned *p_reclen* against the currently-allocated size of the structure (*bufsize*).

This example starts off by allocating an extra **SDIRS_INIT_SIZE** bytes (an arbitrary value) for the stripe directory pathnames. If *p_reclen* is larger than the size of the structure, this example uses **realloc()** to enlarge the structure and calls **statpfs()** again. It then uses the **NEXTPATH()** macro to step through the *p_sdirs* field of the *statpfs* structure, as discussed earlier for **getpfsinfo()**.

Controlling 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/ioN/rmtX	Raw cartridge tape, rewinds automatically when closed.
/dev/ioN/nrmtX	Raw cartridge tape, does not rewind automatically when closed.
/dev/ioN/rmtcX	Raw cartridge tape with compression, rewinds automatically when closed.
/dev/ioN/nrmtcX	Raw cartridge tape with compression, does not rewind automatically when closed.

NOTE

The *rmtc* devices can only be used with tape drives that support data compression.

In each case, N is the node number of the I/O node to which the tape device is connected, and X is the SCSI ID of the tape device (typically 6). So, for example, to use the cartridge tape device with SCSI ID 6 on the boot node (node 0) and have it rewind automatically when closed, use the pathname */dev/io0/rmt6*. To use the same device but have it *not* rewind automatically when closed, use the pathname */dev/io0/nrmt6*.

Performing Operations on Tape Devices

When you call **ioctl()** with **MTIOCTOP** as its second argument, you must use a structure of type *mtop* as the third argument. 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/io0/rmt6*:

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

main() {
    int fd;
    struct mtop s;

    fd = open("/dev/io0/rmt6", O_RDONLY, 0666);
    if(fd == -1) {
        perror("opening /dev/io0/rmt6");
        exit(1);
    }

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

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/io0/rmt6*:

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

main() {
    int fd;
    struct mtget s;

    fd = open("/dev/io0/rmt6", O_RDONLY, 0666);
    if(fd == -1) {
        perror("opening /dev/io0/rmt6");
        exit(1);
    }

    if (ioctl(fd, MTIOCGET, &s) == -1) {
        perror("getting status of tape");
        exit(2);
    }

    printf("mt_type  = 0x%x\n", s.mt_type);
    printf("mt_dsreg = 0x%x\n", s.mt_dsreg);
    printf("mt_erreg = 0x%x\n", s.mt_erreg);
    printf("mt_resid = 0x%x\n", s.mt_resid);
}
```

Synchronization Summary

Table 5-4 lists the I/O modes and summarizes the I/O calls that are synchronizing calls in each one. Table 5-5 lists the most commonly-used I/O calls and summarizes the I/O modes that cause them to become synchronizing calls.

Table 5-4. Synchronization in Each I/O Mode

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

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

Call	I/O Modes Causing the Call to Synchronize
close()	**M_LOG**, **M_SYNC**, **M_RECORD**, and **M_GLOBAL**
cread() and **creadv**()	**M_SYNC** and **M_GLOBAL**
cwrite() and **cwritev**()	**M_SYNC** and **M_GLOBAL**
eseek()	**M_SYNC**, **M_RECORD**, and **M_GLOBAL**
gopen()	All
iread() and **ireadv**()	**M_SYNC** and **M_GLOBAL**
iseof()	**M_SYNC** and **M_GLOBAL**
iwrite() and **iwritev**()	**M_SYNC** and **M_GLOBAL**
lseek()	**M_SYNC**, **M_RECORD**, and **M_GLOBAL**
setiomode()	All

I/O Performance

If your application performs I/O to files, you can use the following techniques to improve its I/O performance. Note: the term *request size* refers to the number of bytes specified in a single **read** or **write** operation. Techniques discussed in this section include:

- Use PFS file systems.
- Use **gopen()** instead of **open()**.
- Use parallel I/O calls.
- Use asynchronous calls.
- Use the appropriate I/O mode.
- Align I/O buffers with virtual memory pages.
- Read or write whole file system blocks.
- Make good use of file striping.

See Chapter 5 for more information on these techniques.

Use PFS File Systems

Always store large data files in file systems of type *PFS* (*Parallel File System*). These file systems are optimized for large I/O requests (request sizes of 64K bytes or more) and simultaneous access by multiple nodes, and files in them can be larger than 2G bytes in size.

Use gopen() Instead of open()

If all nodes in an application open the same file, you should always use **gopen()** rather than **open()**. If all nodes call **open()**, each node sends an "open file" message to the same I/O node at the same time, which can swamp the I/O node with messages. But when all nodes call **gopen()**, only one node communicates with the I/O node; the open file descriptor is then broadcast to the other nodes in the application through efficient global communication techniques. If you *must* use **open()**, try to keep all the nodes from calling it at the same time (do *not* precede the **open()** with a **gsync()**).

Use Parallel I/O Calls

If you program in Fortran, you should always use Paragon OSF/1 *parallel I/O calls*, such as **cread()**, to access your files. These calls give much better performance than the standard Fortran file I/O statements, such as **READ**.

If you program in C, you will not see any I/O performance increase from using parallel I/O calls, such as **cread()**, rather than standard UNIX I/O calls, such as **read()** (although **cread()** gives better performance than **fread()**, which is the C equivalent of Fortran's **READ**). However, you may be able to improve *computational* performance by using asynchronous I/O calls.

Use Asynchronous Calls

The parallel I/O calls are available in both *synchronous* versions (call names beginning with **c**) and *asynchronous* versions (call names beginning with **i**). Synchronous calls do not return until the I/O operation is complete; asynchronous calls return immediately, giving you an I/O ID that you can use to check when the operation is complete.

Although the synchronous calls are easier to use and have slightly lower overhead, you should use the asynchronous calls whenever the results of the call are not needed immediately. Using asynchronous calls can let your application do useful computation in the time when it would otherwise just be waiting for a large I/O operation to complete.

Use the Appropriate I/O Mode

When you use Paragon OSF/1 parallel I/O calls, you can choose from five I/O modes (**M_UNIX**, **M_LOG**, **M_SYNC**, **M_RECORD**, and **M_GLOBAL**), each of which is optimized for a particular pattern of file I/O. Be sure to use the correct I/O mode for your application's usage. In particular:

- Don't use **M_UNIX**, the default I/O mode, unless your application depends on its semantics.
- If all nodes read the same data from the same file at the same time, use **M_GLOBAL**.
- If all nodes read or write the same file, but each node is accessing a different part of the file, use **M_RECORD** if at all possible. This mode provides much higher multi-node performance than the other modes, all of which force reads and writes from different nodes to the same file to be performed in strict sequential order (this is required to preserve standard UNIX I/O semantics, but slows the application down).
- If **M_RECORD** cannot be used because the I/O request size is not constant across all compute nodes, use **M_SYNC** instead.

Align I/O Buffers with Virtual Memory Pages

Try to ensure that memory buffers used in I/O calls are aligned on an 8K-byte boundary whenever possible, to align with the operating system's virtual memory page size. This alignment is particularly important in scatter/gather operations with large request sizes to multiple I/O nodes. If you do not specify properly-aligned buffers, the software must copy the data to new, aligned buffers, which decreases performance.

To be sure the buffer is well-aligned, you should use **malloc()** to allocate it rather than allocating it statically. (This call is available in both Fortran and C.) Buffers allocated with the standard **malloc()** or its derivatives will always be aligned on a 32-byte boundary. See "Align Application Buffers" on page 8-12 for more information on aligning buffers.

Read or Write Whole File System Blocks

Disk space is allocated and managed in units called *file system blocks*. The size of each block in a file system is determined when the file system is created. For best performance, PFS file systems should have a file system block size of 64K bytes.

The file system block size is important because files always begin at a block boundary and data is most efficiently transferred to and from the physical disk in integer numbers of blocks. Furthermore, if a block is modified (but not entirely overwritten) by a write operation, the block may have to be read, modified in memory, and then written back.

Because of this, you will get the best I/O performance if each read or write request begins on a block boundary (a multiple of the block size from the beginning of the file) and the request size is a multiple of the file system block size.

To determine the block size of a file system, you can use the **statfs()** or **fstatfs()** call (see the *OSF/1 Programmer's Reference* for information on these calls), or ask your system administrator.

Make Good Use of File Striping

Files in PFS file systems are distributed, or *striped*, across several directories called *stripe directories*. The number of stripe directories in a PFS file system is called the *stripe factor*, and the amount of data from each file that is stored in each directory is called the *stripe unit*. The product of the stripe factor and the stripe unit is called the *full stripe size*. A PFS file system's stripe factor and stripe unit are set by the system administrator when the PFS file system is mounted.

Each stripe directory is typically on a separate disk, and each disk is typically controlled by a separate I/O node; you get the best I/O performance when you keep all the I/O nodes busy at once. You can use file striping to help you do this, with two different methods:

1. Use a request size equal to an integer multiple of the full stripe size, and make the starting address of each request the beginning of a full stripe. With this method, each I/O request goes to all the I/O nodes at once. This method can be used on any number of nodes.

2. Use a request size equal to the stripe unit size, make the starting address of each request the beginning of a stripe unit, and choose the starting address of each node's requests so that the nodes' requests are evenly distributed among the I/O nodes. With this method, each I/O request goes to just one I/O node, but the application's I/O requests are distributed among the I/O nodes. This method should be used only if the number of compute nodes is greater than or equal to the number of I/O nodes, preferably an integer multiple of the number of I/O nodes.

These two methods are illustrated in Figure 8-1. Note that method 1 uses fewer, larger requests and method 2 uses more, smaller requests. Method 1 is generally more efficient, but method 2 may give better performance for some situations (depending on the number of compute nodes, the number of I/O nodes, the amount of memory on each I/O node, and the size and frequency of requests). If possible, you should try both methods and use whichever is more efficient. The example program in */usr/share/examples/c/stripe* demonstrates the two methods, and you can use it to help you determine which method is best for your application. (Note that you will see more consistent results from one run to the next if the data size is large—8M bytes per node or more.)

You should always use the I/O mode **M_RECORD** when using these methods. **M_RECORD** is the most efficient I/O mode for this type of I/O, and automatically enforces the distribution of data among the I/O nodes. If you use **M_RECORD**, no file pointer calculation or seeking is required. For example:

```
fd = gopen(file, O_WRONLY|O_CREAT|O_TRUNC, M_RECORD, 0666);

while(data < end) {
    cwrite(fd, data, request_size);
    data += request_size;
}
```

Using this code, if *request_size* is equal to the full stripe size, each compute node automatically accesses all I/O nodes on each write (method 1). Alternatively, if *request_size* is equal to the stripe unit, each compute node automatically accesses exactly one I/O node on each write (method 2).

To determine the stripe factor and stripe unit of a PFS file system, you can use the **showfs** command (described under "Displaying File System Attributes" on page 5-5) or the **statpfs()** or **fstatpfs()** call (available only in C; described under "Getting Information About PFS File Systems" on page 5-39). The example program in */usr/share/examples/c/stripe* shows you how you can do this with **fstatpfs()**.

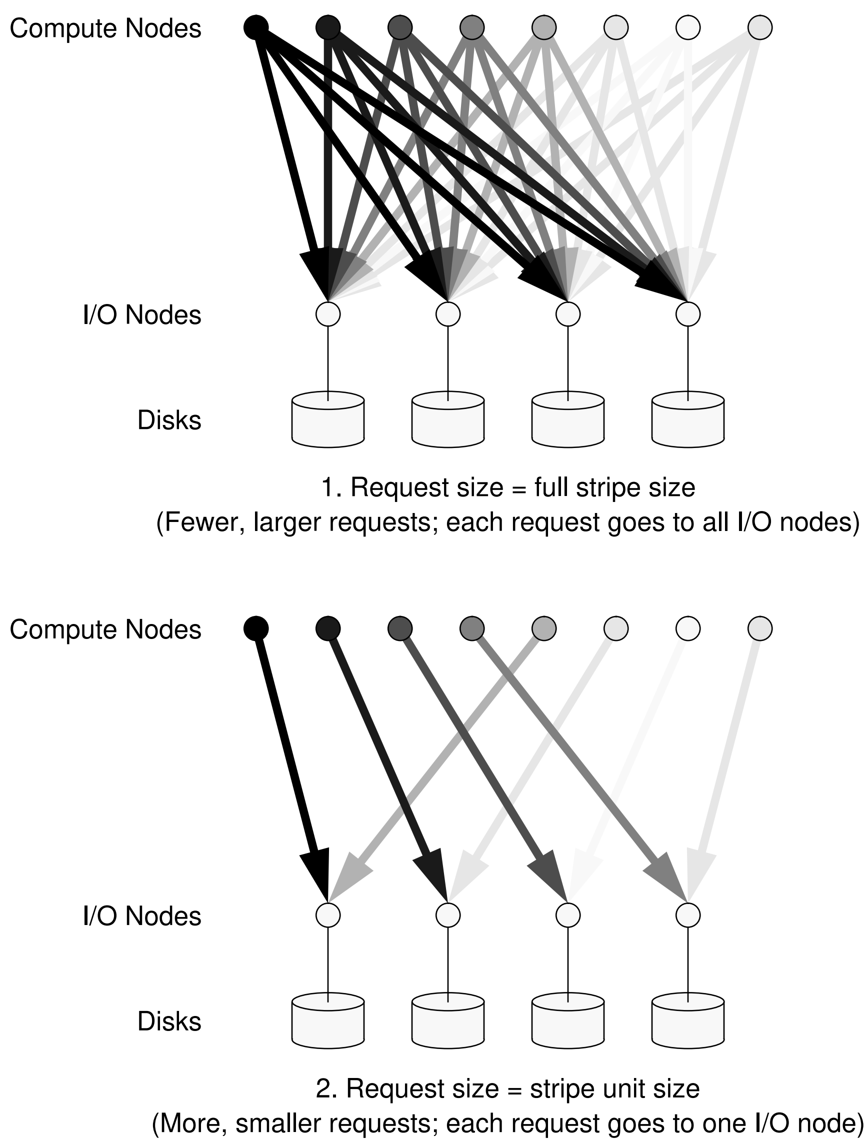

Figure 8-1. Two Methods of Improving I/O Performance with M_RECORD