Chapter 13

Parallel I/O

Rajeev Thakur & William Gropp

Many parallel applications need to access large amounts of data. In such applications, the I/O performance can play a significant role in the overall time to completion. Although I/O is always much slower than computation, it is still possible to achieve good I/O performance in parallel applications by using a combination of sufficient amount of high-speed I/O hardware, appropriate file-system software, appropriate API for I/O, a high-performance implementation of the API, and by using that API the right way. We explain these points in further detail in this chapter.

We begin by explaining what parallel I/O means, how it arises, and why it is a problem. We give an overview of the infrastructure that currently exists for parallel I/O on modern parallel systems, including I/O architecture, parallel file systems, high-level libraries, and application programming interfaces (APIs) for parallel I/O. We explain how the API plays a key role in enabling (or preventing) high performance and how the lack of an appropriate standard API for parallel I/O has hindered performance and portability.

Much of the research in parallel I/O over the last several years has contributed to the definition of the new standard API for parallel I/O that is part of the MPI-2 standard [31]. We discuss the evolution and emergence of this API, often just called MPI-IO, and introduce it with a simple example program. We also describe some optimizations enabled by MPI-IO that are critical for high performance. Finally, we provide guidelines on what users can do to achieve high I/O performance in their applications.

Our focus is mainly on the type of parallel I/O commonly seen in high-end scientific computing and not on the I/O that arises in databases, transaction processing, and other commercial applications. I/O in parallel scientific computing often involves large data objects, such as a single large array, distributed

across hundreds of processors. In contrast, while the amount of data stored and accessed in a commercial database may be larger than the data stored as a result of a scientific simulation, each record in a commercial database is usually very small.

13.1 Introduction

Any application, sequential or parallel, may need to access data stored in files for many reasons, such as reading the initial input, writing the results, checkpointing for later restart, data analysis, and visualization [18]. In this chapter we are concerned mainly with *parallel* applications consisting of multiple processes (or threads[1]) that need to access data stored in files. We define parallel I/O as concurrent requests from multiple processes of a parallel program for data stored in files. Accordingly, at least two scenarios are possible:

- Each process accesses a separate file; that is, no file is shared among processes, or
- All processes access a single, shared file.

While the former scenario can be considered as parallel I/O in some sense because it represents I/O performed by a parallel program, it is actually just sequential (uniprocess) I/O performed independently by a number of processes. The latter case, where all processes access a shared file, is true parallel I/O and represents what the term "parallel I/O" means as used in this chapter. In other words, the I/O is parallel from the application's perspective.

In recent years, although great advances have been made in the CPU and communication performance of parallel machines, similar advances have not been made in their I/O performance. The densities and capacities of disks have increased significantly, but improvement in performance of individual disks has not followed the same pace. Although parallel machines with peak performance of 1 Tflops/sec or more are available, applications running on parallel machines usually achieve I/O bandwidths of at most a few hundred Mbytes/sec. In fact, many applications achieve less than 10 Mbytes/sec [12].

As parallel computers get bigger and faster, scientists are increasingly using them to solve problems that not only need a large amount of computing power but also need to access large amounts of data. (See [14, 26, 38] for a list of many such applications.) Since I/O is slow, the I/O speed, and not the CPU or communication speed, is often the bottleneck in such applications. For parallel computers to be truly usable for solving real, large-scale problems, the I/O performance must be scalable and balanced with respect to the CPU and communication performance of the system.

The rest of this chapter is organized as follows. In Section 13.2 we describe the existing infrastructure for parallel I/O, including architecture, file systems,

[1]The discussion in this chapter refers to multiple processes rather than threads because our focus is on the MPI-IO model for parallel I/O. Nonetheless, the issues we discuss apply equally well to a parallel-programming model based on multiple threads within a process.

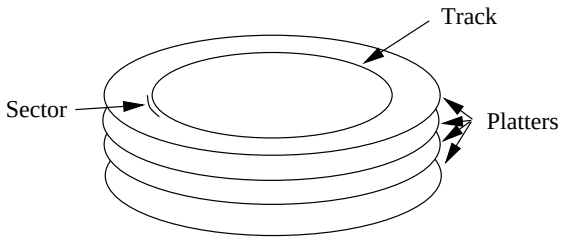

Figure 13.1: Schematic of a typical disk

and high-level libraries. We also discuss the issue of application programming interfaces (APIs) for parallel I/O and explain how the lack of an appropriate standard API has hindered performance and portability in the past. In Section 13.3 we introduce the new MPI-IO standard API, which has the potential to solve the API problem and deliver performance and portability. In Section 13.4 we describe some optimizations that are critical to parallel I/O performance. In Section 13.5 we provide some guidelines on how users can achieve high I/O performance in their applications. We summarize the chapter in Section 13.6.

13.2 Parallel I/O Infrastructure

In this section we give a brief overview of the infrastructure for parallel I/O that currently exists on parallel machines. We begin by reviewing basic, nonparallel I/O.

13.2.1 Basic Disk Architecture

The most common secondary-storage device is a *disk*. A disk consists of one or more *platters* coated with a magnetic medium. The disk spins at a relatively high rate; 5,000–10,000 RPM (revolutions per minute) are common. A platter is divided into a number of concentric *tracks*, which are themselves divided into smaller arcs called *sectors*. A sector is the smallest addressable unit on the disk, and a typical sector size is 512 bytes [61]. Data is read by one or more *heads* that can move across the platters. A schematic of a disk is shown in Figure 13.1.

Data from a disk is typically accessed in multiples of sectors stored contiguously, sometimes called a *cluster*. On commodity disks, a minimum of 32 sectors (16 Kbytes) or more are accessed in a single operation. As a result, reading or writing a single byte of data from or to a disk actually causes thousands of bytes to be moved. In other words, there can be a huge difference between the amount of data logically accessed by an application and the amount of data physically moved, as demonstrated in [49]. In addition, a substantial latency is introduced by the need to wait for the right sector to move under a read or write head—even at 10,000 RPM, it takes 6 milliseconds for the disk to complete one revolution. To avoid accessing the disk for each I/O request, an operating system typically maintains a cache in main memory, called the *file-system cache*, that contains parts of the disk that have been recently accessed. Data written

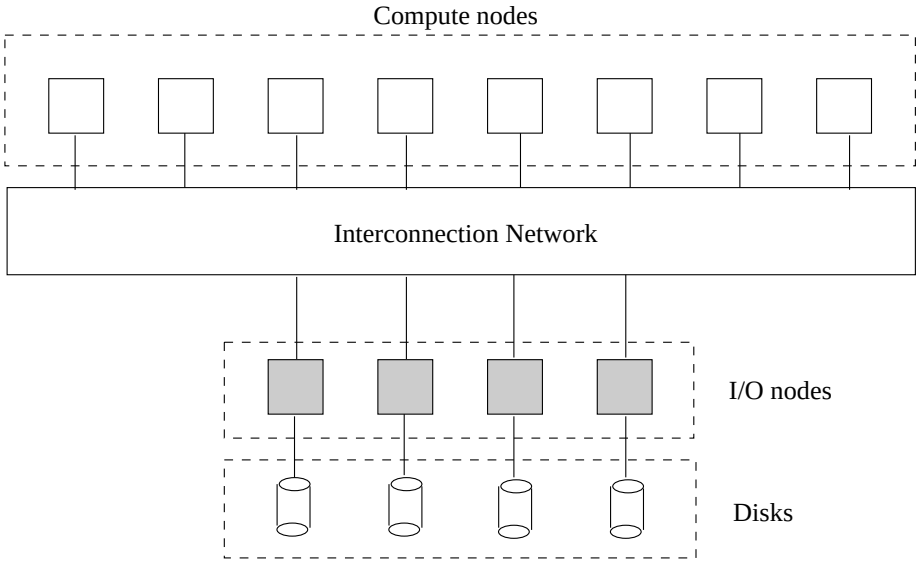

Figure 13.2: General parallel I/O architecture of distributed-memory systems

to the cache is periodically flushed to the disk by an operating-system daemon. Despite the cache, it is inefficient to read or write small amounts of data from an application. Applications that need high I/O performance must ensure that all I/O operations access large amounts of data.

Further details about disk architecture can be found in [7, 61].

13.2.2 Parallel I/O Architecture

Let us now consider the I/O architectures of parallel machines. We first consider distributed-memory machines, examples of which include the IBM SP, ASCI Red (Intel Tflops), Cray T3E, clusters of workstations, and older machines such as the Thinking Machines CM-5 and Intel Paragon and iPSC hypercubes. Figure 13.2 shows the general I/O architecture of a distributed-memory machine. In addition to the compute nodes, the machine has a set of I/O nodes. The I/O nodes are connected to each other and to the compute nodes usually by the same interconnection network that connects the compute nodes. Each I/O node is connected to one or more storage devices, each of which could be either an individual disk or an array of disks, such as a RAID (Redundant Array of Inexpensive Disks) [7, 40]. The I/O nodes function as servers for the parallel file system. The parallel file system typically stripes files across the I/O nodes and disks by dividing the file into a number of smaller units called *striping units* and assigning the striping units to disks in a round-robin manner. File striping provides higher bandwidth and enables multiple compute nodes to access distinct portions of a file concurrently.

Usually, but not always, the I/O nodes are dedicated for I/O and no compute

jobs are run on them. On many machines, each of the compute nodes also has a local disk of its own, which is usually not directly accessible from other nodes. These disks are not part of the common "parallel I/O system" but are used to store scratch files local to each process and other files used by the operating system.

This kind of architecture allows concurrent requests from multiple compute nodes to be serviced simultaneously. Parallelism comes about in multiple ways: parallel data paths from the compute nodes to the I/O nodes, multiple I/O nodes and file-system servers, and multiple storage devices (disks). If each storage device is a disk array, it provides even more parallelism.

Shared-memory machines typically do not have this kind of I/O architecture; they do not have separate I/O nodes. Examples of such machines are the SGI Origin2000, Cray T90, HP Exemplar, and NEC SX-4. On these machines, the operating system schedules the file-system server on the compute nodes. Nonetheless, these machines can be configured with multiple disks, and the file system can stripe files across the disks. The disks are connected to the machine via SCSI or Fibre Channel connections, just as they are in distributed memory machines.

For further information on parallel I/O architecture we refer readers to the excellent surveys in [15] and [27].

A relatively new area of research is that of network-attached storage devices (NASD) [19]. In NASD, storage devices are not directly connected to their host systems via a specialized I/O bus, but instead communicate with their host systems through a high-performance network such as Fibre Channel [16]. This approach has the potential to improve performance and scalability by providing direct data transfer between client and storage and eliminating the server, which can be a bottleneck.

13.2.3 File Systems

A number of commercial and research file systems have been developed over the last few years to meet the needs of parallel I/O. We briefly describe some of them below and provide pointers to additional information.

One of the first commercial parallel file systems was the Intel Concurrent File System (CFS) for the Intel iPSC hypercubes. It had a Unix-like API with the addition of various file-pointer modes [42]. CFS evolved into the Parallel File System (PFS) on the Intel Paragon, but retained the same API. The CM-5, nCUBE, and Meiko CS-2 also had their own parallel file systems [15]. A different API was introduced by the Vesta file system, developed at the IBM Watson Research Center [10]. Vesta provided the initial parallel file system for the IBM SP. The unique feature of Vesta was that it supported logical file views and noncontiguous file accesses—a departure from the traditional Unix API. Vesta evolved into an IBM product called PIOFS, which remained the parallel file system on the SP until recently. The current parallel file system on the IBM SP is called GPFS [1], which, interestingly, is not backward compatible with PIOFS. It does not support PIOFS file views or noncontiguous

file accesses; instead, it supports the POSIX I/O interface [24]. However, for noncontiguous accesses, users can use the MPI-IO interface on top of GPFS by using either IBM's implementation of MPI-IO or other implementations, such as ROMIO [45]. Unlike other parallel file systems, GPFS follows a shared-disk model rather than a client-server model [1]. Shared-memory multiprocessors also have high-performance file systems that allow concurrent access to files. Examples of such file systems are XFS on the SGI Origin2000, HFS on the HP Exemplar, and SFS on the NEC SX-4. Sun has developed a parallel file system, Sun PFS, for clusters of Sun SMPs [66].

A number of parallel file systems have also been developed by various research groups. The Galley parallel file system developed at Dartmouth College supports a three-dimensional file structure consisting of files, subfiles, and forks [35]. PPFS is a parallel file system developed at the University of Illinois for clusters of workstations [23]. The developers use it as a testbed for research on various aspects of file-system design, such as caching/prefetching policies and automatic/adaptive policy selection [29, 30]. PVFS is a parallel file system for Linux clusters developed at Clemson University [64]. PVFS stripes files across the local disks of machines in a Linux cluster and provides the look-and-feel of a single Unix file system. The regular Unix commands, such as `rm`, `ls`, and `mv`, can be used on PVFS files, and the files can be accessed from a (parallel) program by using the regular Unix I/O functions. PVFS is also packaged in a way that makes it very easy to download, install, and use.

Distributed/networked file systems are a rich area of research. Examples of such file systems are xFS [2], AFS/Coda [8], and GFS [60]. We do not discuss them in this chapter; interested readers can find further information in the papers cited above.

13.2.4 The API Problem

Most commercial parallel file systems have evolved out of uniprocessor file systems, and they retain the same API, namely, the Unix I/O API. The Unix API, however, is not an appropriate API for parallel I/O for two main reasons: it does not allow noncontiguous file accesses and it does not support collective I/O. We explain these reasons below.

The Unix read/write functions allow users to access only a single contiguous piece of data at a time.[2] While such an API may be sufficient for the needs of uniprocess programs, it is not sufficient for the kinds of access patterns common in parallel programs. Many studies of the I/O access patterns in parallel programs have shown that each process of a parallel program may need to access several relatively small, noncontiguous pieces of data from a file [3, 12, 36, 50, 51, 56]. In addition, many/all processes may need to access the file at about the same time, and, although the accesses of each process may

[2]Unix does have functions `readv` and `writev`, but they allow noncontiguity only in memory and not in the file. POSIX has a function `lio_listio` that allows users to specify a list of requests at a time, but each request is treated internally as a separate asynchronous I/O request, the requests can be a mixture of reads and writes, and the interface is not collective.

be small and noncontiguous, the accesses of different processes may be interleaved in the file and together may span large contiguous chunks. Such access patterns occur because of the manner in which data stored in a shared file is distributed among processes. With the Unix I/O interface, the programmer has no means of conveying this "big picture" of the access pattern to the I/O system. Each process must seek to a particular location in the file, read or write a small contiguous piece, then seek to the start of the next contiguous piece, read or write that piece, and so on. The result is that each process makes hundreds or thousands of requests for small amounts of data. Numerous small I/O requests arriving in any order from multiple processes results in very poor performance, not just because I/O latency is high but also because the file-system cache gets poorly utilized.

The example in Figure 13.3 illustrates this point. The figure shows an access pattern commonly found in parallel applications, namely, distributed-array access. A two-dimensional array is distributed among 16 processes in a (block, block) fashion. The array is stored in a file corresponding to the global array in row-major order, and each process needs to read its local array from the file. The data distribution among processes and the array storage order in the file are such that the file contains the first row of the local array of process 0, followed by the first row of the local array of process 1, the first row of the local array of process 2, the first row of the local array of process 3, then the second row of the local array of process 0, the second row of the local array of process 1, and so on. In other words, the local array of each process is not located contiguously in the file. To read its local array with a Unix-like API, each process must seek to the appropriate location in the file, read one row, seek to the next row, read that row, and so on. Each process must make as many read requests as the number of rows in its local array. If the array is large, the file system may receive thousands of read requests.

Instead, if the I/O API allows the user to convey the entire access information of each process as well as the fact that all processes need to access the file simultaneously, the implementation (of the API) can read the entire file contiguously and simply send the right pieces of data to the right processes. This optimization, known as collective I/O, can improve performance significantly [13, 28, 48, 58]. The I/O API thus plays a critical role in enabling the user to express I/O operations conveniently and also in conveying sufficient information about access patterns to the I/O system so that the system can perform I/O efficiently.

Another problem with commercial parallel-file-system APIs is the lack of portability. Although parallel file systems have Unix-like APIs, many vendors support variations of the Unix (or POSIX [24]) API, and, consequently, programs written with these APIs are not portable.

13.2.5 I/O Libraries

A number of I/O libraries have also been developed over the last several years, mostly as part of research projects. These libraries either provide a better API

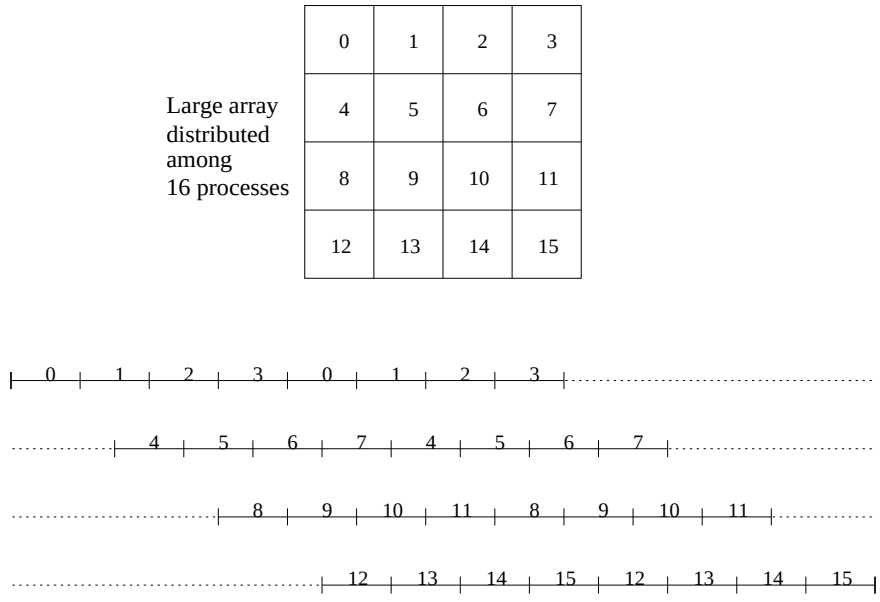

Figure 13.3: Common access pattern in parallel applications: distributed-array access. The numbers on the line indicate the process that needs a particular portion of the file.

than Unix I/O and perform I/O optimizations enabled by the API or provide some convenience features useful to applications that file systems do not provide. We list some of these libraries below.

The PASSION library, developed at Syracuse University, supports efficient to arrays and sections of arrays stored in files [55]. It uses data sieving, two-phase collective I/O, and (recently) compression as the main optimizations. The Panda library, developed at the University of Illinois, also supports high-performance array access [48]. It uses server-directed collective I/O and chunked storage as the main optimizations. SOLAR is a library for out-of-core linear-algebra operations, developed at IBM Watson Research Center [65]. The ChemIO library, developed at Pacific Northwest National Laboratory, provides I/O support for computational-chemistry applications [34].

HDF [63], netCDF [33], and DMF [47] are libraries designed to provide even higher level of I/O support to applications. For example, they can directly read/write meshes and grids. Such libraries also try to hide I/O parallelism from the application, often to the detriment of performance. Nonetheless, these libraries are very popular among application developers because they provide a level of abstraction that application developers need.

Because all the libraries mentioned above support their own API, usually much different from the Unix I/O API, they do not solve the API portability

problem.

13.2.6 Language-Based Parallel I/O

Some efforts have been made to support parallel I/O directly in the parallel programming language. For example, the Fortran D and Fortran 90D research projects explored the use of language-based parallel I/O with a combination of compiler directives and runtime library calls [4, 5, 39]. CM Fortran from Thinking Machines Corp. also supported reading and writing of parallel arrays. Although parallel I/O was discussed during the deliberations of the High Performance Fortran (HPF) Forum, it does not appear in the final HPF standard. In all, language-based parallel I/O remains mainly a research effort.

13.2.7 Need for a Standard I/O API

Although great strides were made in parallel I/O research in the early 1990s, there remained a critical need for a single, standard, language-neutral API designed specifically for parallel I/O performance and portability. Fortunately, such an API now exists. It is the I/O interface defined as part of the MPI-2 standard, often referred to as MPI-IO [21, 31].

13.3 Overview of MPI-IO

In this section we give a brief overview of MPI-IO, describe its main features, and elaborate on one important feature—the ability to specify noncontiguous I/O requests by using MPI's derived datatypes.

13.3.1 Background

MPI-IO originated in an effort begun in 1994 at IBM Watson Research Center to investigate the impact of the (then) new MPI message-passing standard on parallel I/O. A group at IBM wrote an important paper [44] that explores the analogy between MPI message passing and I/O. Roughly speaking, one can consider reads and writes to a file system as receives and sends of messages. This paper was the starting point of MPI-IO in that it was the first attempt to exploit this analogy by applying the (then relatively new) MPI concepts for message passing to the realm of parallel I/O.

The idea of using message-passing concepts in an I/O library appeared successful, and the effort was expanded into a collaboration with parallel I/O researchers from NASA Ames Research Center. The resulting specification appeared in [9]. At this point a large email discussion group was formed, with participation from a wide variety of institutions. This group, calling itself the MPI-IO Committee, pushed the idea further in a series of proposals, culminating in [62].

During this time, the MPI Forum had resumed meeting to address a number of topics that had been deliberately left out of the original MPI Standard, including parallel I/O. The MPI Forum initially recognized that both the MPI-IO Committee and the Scalable I/O Initiative [46] represented efforts to develop

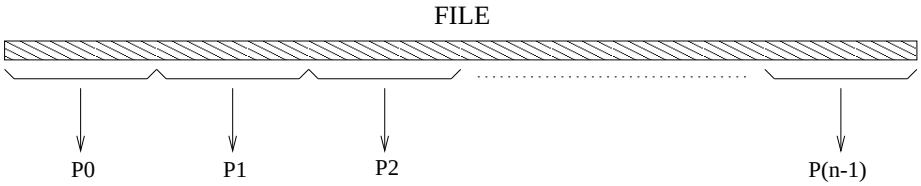

Figure 13.4: Each process needs to read a chunk of data from a common file

a standard parallel I/O interface and therefore decided not to address I/O in its deliberations. In the long run, however, the three threads of development—by the MPI-IO Committee, the Scalable I/O Initiative, and the MPI Forum—merged because of a number of considerations. The result was that, from the summer of 1996, the MPI-IO design activities took place in the context of the MPI Forum meetings. The MPI Forum used the latest version of the existing MPI-IO specification [62] as a starting point for the I/O chapter in MPI-2. The I/O chapter evolved over many meetings of the Forum and was released in its final form along with the rest of MPI-2 in July 1997 [31]. MPI-IO now refers to this I/O chapter in MPI-2.

13.3.2 Simple MPI-IO Example

To get a flavor of what MPI-IO looks like, let us consider a simple example: a parallel program in which processes need to read data from a common file. Let us assume that there are n processes, each needing to read $(1/n)$th of the file as shown in Figure 13.4. Figure 13.5 shows one way of writing such a program with MPI-IO. It has the usual functions one would expect for I/O: an open, a seek, a read, and a close. Let us look at each of the functions closely.

`MPI_File_open` is the function for opening a file. The first argument to this function is a communicator that indicates the group of processes that need to access the file and that are calling this function. This communicator also represents the group of processes that will participate in any collective I/O operations on the open file. In this simple example, however, we don't use collective I/O functions. We pass `MPI_COMM_WORLD` as the communicator, meaning that all processes need to open and thereafter access the file. The file name is passed as the second argument to `MPI_File_open`. The third argument to `MPI_File_open` specifies the mode of access; we use `MPI_MODE_RDONLY` because this program only reads from the file. The fourth argument, called the *info* argument, allows the user to pass hints to the implementation. In this example, we don't pass any hints; instead, we pass a null info argument, `MPI_INFO_NULL`. `MPI_File_open` returns a *file handle* in the last argument. This file handle is to be used for future operations on the open file.

After opening the file, each process moves its local file pointer, called an individual file pointer, to the location in the file from which the process needs to read data. We use the function `MPI_File_seek` for this purpose. The first argument to `MPI_File_seek` is the file handle returned by `MPI_File_open`. The

```
/* read from a common file using individual file pointers */
#include "mpi.h"

#define FILESIZE (1024 * 1024)

int main(int argc, char **argv)
{
    int *buf, rank, nprocs, nints, bufsize;
    MPI_File fh;
    MPI_Status status;

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

    bufsize = FILESIZE/nprocs;
    buf = (int *) malloc(bufsize);
    nints = bufsize/sizeof(int);

    MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
                  MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);
    MPI_File_seek(fh, rank*bufsize, MPI_SEEK_SET);
    MPI_File_read(fh, buf, nints, MPI_INT, &status);
    MPI_File_close(&fh);

    free(buf);
    MPI_Finalize();
    return 0;
}
```

Figure 13.5: Simple MPI-IO program to perform the I/O needed in Figure 13.4

second argument specifies the offset in the file to seek to, and the third argument `MPI_SEEK_SET` specifies that the offset must be calculated from the head of the file. We specify the offset to `MPI_File_seek` as a product of the rank of the process and the amount of data to be read by each process.

We use the function `MPI_File_read` for reading data. On each process, this function reads data from the current location of the process's individual file pointer for the open file. The first argument to `MPI_File_read` is the file handle. The second argument is the address of the buffer in memory into which data must be read. The next two arguments specify the amount of data to be read. Since the data is of type integer, we specify it as a count of the number of integers to be read. The final argument is a status argument, which is the same as the status argument in MPI communication functions, such as

MPI_Recv. One can determine the amount of data actually read by using the functions MPI_Get_count or MPI_Get_elements on the status object returned by MPI_File_read, but we don't bother to do so in this example. MPI_File_read increments the individual file pointer on each process by the amount of data read by that process. Finally, we close the file using the function MPI_File_close.

The five functions, MPI_File_open, MPI_File_seek, MPI_File_read, MPI_File_write, and MPI_File_close, are actually sufficient to write any I/O program. The other MPI-IO functions are for performance, portability, and convenience. Although these five functions[3] can be used as a quick start to using MPI-IO and for easily porting Unix I/O programs to MPI-IO, users must not stop here. For real benefits with using MPI-IO, users must use its special features, such as support for noncontiguous accesses and collective I/O. This issue is discussed further in Section 13.5 and in [21].

13.3.3 Main Features of MPI-IO

MPI-IO is a rich interface with many features specifically intended for portable, high-performance parallel I/O. It has bindings in three languages: C, Fortran, and C++.

MPI-IO supports three kinds of basic data-access functions: using an explicit offset, individual file pointer, and shared file pointer. The explicit-offset functions take as argument the offset in the file from which the read/write should begin. The individual-file-pointer functions read/write data from the current location of a file pointer that is local to each process. The shared-file-pointer functions read/write data from the location specified by a common file pointer shared by the group of processes that together opened the file. In all these functions, users can specify a noncontiguous data layout in memory and file. Both blocking and nonblocking versions of these functions exist. MPI-IO also has collective versions of these functions, which must be called by all processes that together opened the file. The collective functions enable an implementation to perform collective I/O. A restricted form of nonblocking collective I/O, called split collective I/O, is supported.

A unique feature of MPI-IO is that it supports multiple data-storage representations: native, internal, external32, and also user-defined representations. native means that data is stored in the file as it is in memory; no data conversion is performed. internal is an implementation-defined data representation that may provide some (implementation-defined) degree of file portability. external32 is a specific, portable data representation defined in MPI-IO. A file written in external32 format on one machine is guaranteed to be readable on any machine with any MPI-IO implementation. MPI-IO also includes a mechanism for users to define a new data representation by providing data-conversion functions, which MPI-IO uses to convert data from file format to memory format

[3] The reader familiar with threads will note that the seek operation is not thread-safe: it effectively sets a global variable (the position in the file) that another thread could change before the subsequent read or write operation. MPI-IO has thread-safe variants of MPI_File_read and MPI_File_write, called MPI_File_read_at and MPI_File_write_at, that combine the seek and read/write operation.

and vice versa.

MPI-IO provides a mechanism, called *info*, that enables users to pass hints to the implementation in a portable and extensible manner. Examples of hints include parameters for file striping, prefetching/caching information, and access-pattern information. Hints do not affect the semantics of a program, but they may enable the MPI-IO implementation or underlying file system to improve performance or minimize the use of system resources [6, 41].

MPI-IO also has a set of rigorously defined consistency and atomicity semantics that specify the results of concurrent file accesses.

For details of all these features, we refer readers to [20, 21, 31]. We elaborate further on only one feature—the ability to access noncontiguous data with a single I/O function by using MPI's derived datatypes—because it is critical for high performance in parallel applications. We emphasize this point because achieving high performance requires both a proper API and proper use of that API by the programmer. Other I/O efforts have also addressed the issue of accessing noncontiguous data; one example is the low-level API [11] developed as part of the Scalable I/O Initiative [46]. MPI-IO, however, is the only widely deployed API that supports noncontiguous access.

13.3.4 Noncontiguous Accesses in MPI-IO

In MPI, the amount of data a function sends or receives is specified in terms of instances of a *datatype* [32]. Datatypes in MPI are of two kinds: basic and derived. Basic datatypes are those that correspond to the basic datatypes in the host programming language—integers, floating-point numbers, and so forth. In addition, MPI provides datatype-constructor functions to create derived datatypes consisting of multiple basic datatypes located either contiguously or noncontiguously. The datatype created by a datatype constructor can be used as an input datatype to another datatype constructor. Any noncontiguous data layout can therefore be represented in terms of a derived datatype.

MPI-IO uses MPI datatypes for two purposes: to describe the data layout in the user's buffer in memory and to define the data layout in the file. The data layout in memory is specified by the datatype argument in each read/write function in MPI-IO. The data layout in the file is defined by the *file view*. When the file is first opened, the default file view is the entire file; that is, the entire file is visible to the process, and data will be read/written contiguously starting from the location specified by the read/write function. A process can change its file view at any time by using the function `MPI_File_set_view`, which takes as argument an MPI datatype, called the *filetype*. From then on, data will be read/written only to those parts of the file specified by the filetype; any "holes" will be skipped. The file view and the data layout in memory can be defined by using any MPI datatype; therefore, any general, noncontiguous access pattern can be compactly represented.

13.3.5 MPI-IO Implementations

Several implementations of MPI-IO are available, including portable and vendor-specific implementations. ROMIO is a freely available, portable implementation that we have developed at Argonne [45, 59]. It runs on most parallel computers and networks of workstations and uses the native parallel/high-performance file systems on each machine. It is designed to be used with multiple MPI-1 implementations. Another freely available, portable MPI-IO implementation is PMPIO from NASA Ames Research Center [17, 43]. A group at Lawrence Livermore National Laboratory has implemented MPI-IO on the HPSS mass-storage system [25]. Most vendors either already have an MPI-IO implementation or are actively developing one. SGI and HP have included ROMIO into their MPI product. Sun [66] and Fujitsu have their own (complete) MPI-IO implementations. IBM, Compaq (DEC), NEC, and Hitachi are in various stages of MPI-IO development.

13.4 Parallel I/O Optimizations

In this section we describe some key optimizations in parallel I/O that are critical for high performance. These optimizations include data sieving, collective I/O, and hints and adaptive file-system policies. With the advent of MPI-IO, these optimizations are now supported in the API in a standard, portable way. This in turn enables a library or file system to actually perform these optimizations.

13.4.1 Data Sieving

As mentioned above, in many parallel applications each process may need to access small, noncontiguous pieces of data. Since I/O latency is very high, accessing each contiguous piece separately is very expensive: it involves too many system calls for small amounts of data. Instead, if the user conveys the entire noncontiguous access pattern within a single read or write function, the implementation can perform an optimization called *data sieving* and read or write data with much higher performance. Data sieving was first used in PASSION in the context of accessing sections of out-of-core arrays [53, 55]. We use a very general implementation of data sieving (for any general access pattern) in our MPI-IO implementation, ROMIO. We explain data sieving in the context of its implementation in ROMIO [58].

To reduce the effect of high I/O latency, it is critical to make as few requests to the file system as possible. Data sieving is a technique that enables an implementation to make a few large, contiguous requests to the file system even if the user's request consists of several small, noncontiguous accesses. Figure 13.6 illustrates the basic idea of data sieving. Assume that the user has made a single read request for five noncontiguous pieces of data. Instead of reading each noncontiguous piece separately, ROMIO reads a single contiguous chunk of data starting from the first requested byte up to the last requested byte into a temporary buffer in memory. It then extracts the requested portions from

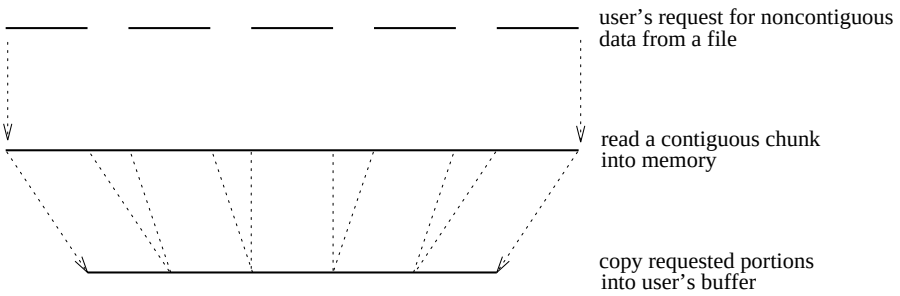

Figure 13.6: Data sieving

the temporary buffer and places them in the user's buffer. The user's buffer happens to be contiguous in this example, but it could well be noncontiguous.

A potential problem with this simple algorithm is its memory requirement. The temporary buffer into which data is first read must be as large as the *extent* of the user's request, where extent is defined as the total number of bytes between the first and last byte requested (including holes). The extent can potentially be very large—much larger than the amount of memory available for the temporary buffer—because the holes (unwanted data) between the requested data segments could be very large. The basic algorithm, therefore, must be modified to make its memory requirement independent of the extent of the user's request.

ROMIO uses a user-controllable parameter that defines the maximum amount of contiguous data that a process can read at a time during data sieving. This value also represents the maximum size of the temporary buffer. The user can change this size at run time via MPI-IO's hints mechanism. If the extent of the user's request is larger than the value of this parameter, ROMIO performs data sieving in parts, reading only as much data at a time as defined by the parameter.

The advantage of data sieving is that data is always accessed in large chunks, although at the cost of reading more data than needed. For many common access patterns, the holes between useful data are not unduly large, and the advantage of accessing large chunks far outweighs the cost of reading extra data. In some access patterns, however, the holes are so large that the cost of reading the extra data outweighs the cost of accessing large chunks. An "intelligent" data-sieving algorithm can handle such cases as well. The algorithm can analyze the user's request and decide whether to perform data sieving or access each contiguous data segment separately. We plan to add this feature to ROMIO.

Data sieving can similarly be used for writing data. A read-modify-write must be performed, however, to avoid destroying the data already present in the holes between contiguous data segments. The portion of the file being accessed must also be locked during the read-modify-write to prevent concurrent updates by other processes. The size of the write buffer can also be changed by the user via hints.

One could argue that most file systems perform data sieving anyway because they perform caching. That is, even if the user makes many small I/O requests, the file system always reads multiples of disk blocks and may also perform a read-ahead. The user's requests, therefore, may be satisfied out of the file-system cache. Our experience, however, has been that the cost of making many system calls, each for small amounts of data, is extremely high, despite the caching performed by the file system. In most cases, it is more efficient to make a few system calls for large amounts of data and extract the needed data.

13.4.2 Collective I/O

In many cases, the data to be read or written represents a single object, distributed across many processors. An example is a single array, distributed across all processes in a parallel application. As we have seen, when this array is written to a file, each process must write many relatively small segments. Yet once the data is in the file, the array is stored in a single, contiguous block in the file. How can we exploit the fact that the entire data to be written fills a large, contiguous block in the file?

If the entire noncontiguous access information of all processes is known, an implementation can optimize the access even further. Instead of reading large chunks and discarding the unwanted data as in data sieving, the unwanted data can be communicated to other processes that need it. Such optimization is broadly referred to as *collective I/O*, and it has been shown to improve performance significantly [13, 28, 48, 58, 66].

Collective I/O can be performed in different ways and has been studied by many researchers in recent years. It can be done at the disk level (disk-directed I/O [28]), at the server level (server-directed I/O [48]), or at the client level (two-phase I/O [13] or collective buffering [37]). Each method has its advantages and disadvantages. Since ROMIO is a portable, user-level library with no separate I/O servers, ROMIO performs collective I/O at the client level using a generalized version of two-phase I/O. We explain the basic concept of two-phase I/O below; details of ROMIO's implementation can be found in [58].

Two-Phase I/O

Two-phase I/O was first proposed in [13] in the context of accessing distributed arrays from files. The basic idea in two-phase I/O is to avoid making lots of small I/O requests by splitting the access into two phases: an I/O phase and a communication phase. Let us consider the example of reading a (block,block) distributed array from a file using two-phase I/O, illustrated in Figure 13.7. In the first phase of two-phase I/O, all processes access data assuming a distribution that results in each process making a single, large, contiguous access. In this example, such a distribution is a row-block or (block,*) distribution. In the second phase, processes redistribute data among themselves to the desired distribution. The advantage of this method is that by making all file accesses large and contiguous, the I/O time is reduced significantly. The added cost of interprocess communication for redistribution is (almost always) small com-

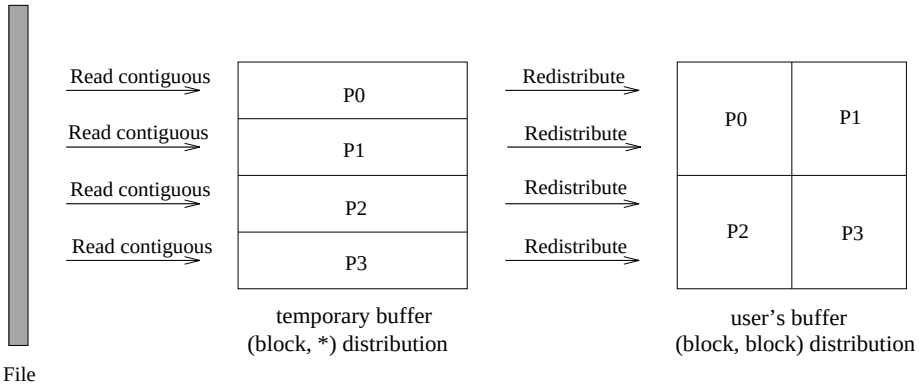

Figure 13.7: Reading a distributed array by using two-phase I/O

pared with the savings in I/O time. The overall performance, therefore, is close to what can be obtained by making large I/O requests in parallel.

The basic two-phase method was extended in [54] to access sections of out-of-core arrays. An even more general version of two-phase I/O is implemented in ROMIO [58]. It supports any access pattern, and the user can also control via hints the amount of temporary memory ROMIO uses as well as the number of processes that actually perform I/O in the I/O phase.

13.4.3 Hints and Adaptive File-System Policies

Parallel applications exhibit such a wide variation in access patterns that any single file-system policy (regarding file-striping parameters, caching/prefetching, etc.) is unlikely to perform well for all applications. Two solutions exist for this problem: either the user can inform the file system (via hints) about the application's access pattern, the desired striping parameters, or the desired caching/prefetching policies, or the file system can be designed to automatically detect and adapt its policies to the access pattern of the application. Various research efforts have demonstrated the benefits of such optimization [6, 29, 30, 41].

As mentioned above, hints can also be used to vary the sizes of temporary buffers used internally by the implementation for various optimizations. Choosing the right buffer size can improve performance considerably, as demonstrated in Section 13.5.2 and in [66].

The hints mechanism in MPI-IO also allows users to specify machine-specific options and optimizations in a portable way. That is, the same program can be run everywhere, and the implementation will simply ignore the hints that are not applicable to the machine on which the program is being run. An example of the use of machine-specific hints are the hints ROMIO accepts for using "direct I/O" on SGI's XFS file system. Direct I/O is an XFS option that can be specified via the `O_DIRECT` flag to the `open` function. In direct I/O, the file system moves data directly between the user's buffer and the storage devices, bypassing the file-system cache and thereby saving an extra copy. Another advantage is that

in direct I/O the file system allows writes from multiple processes and threads to a common file to proceed concurrently rather than serializing them as it does with regular buffered I/O. Direct I/O, however, performs well only if the machine has sufficient I/O hardware for high disk bandwidth. If not, regular buffered I/O through the file-system cache performs better. ROMIO, therefore, does not use direct I/O by default. It uses direct I/O only if the user (who knows whether the machine has high disk bandwidth) recommends it via a hint. On the Argonne Origin2000 configured with 10 Fibre Channel controllers and a total of 100 disks, we obtained bandwidths of around 720 Mbytes/sec for parallel writes and 650 Mbytes/sec for parallel reads with the direct I/O hint specified. Without this hint, the bandwidth was only 100 Mbytes/sec for parallel writes and 300 Mbytes/sec for parallel reads.

Direct I/O can be used only if certain restrictions regarding the memory alignment of the user's buffer, minimum and maximum I/O sizes, alignment of file offset, etc., are met. ROMIO determines whether these restrictions are met for a particular request and only then uses direct I/O; otherwise it uses regular buffered I/O (even if the user specified the direct I/O hint). We plan to add an optimization to ROMIO in which even though the user's request does not meet the restrictions, ROMIO will try to meet the restrictions by reorganizing the data internally, at least in the case of collective I/O routines.

13.5 How Can Users Achieve High I/O Performance?

We provide some general guidelines for achieving high I/O performance and some specific guidelines for achieving high performance with MPI-IO.

13.5.1 General Guidelines

Following are some general guidelines for achieving high I/O performance. Although many of them seem obvious, the reason for poor performance is often that one or more of these simple guidelines are not being followed.

- **Buy Sufficient I/O Hardware for the Machine**: Machines tend to be purchased for high computation and communication performance but are often underconfigured for the I/O requirements of the applications being run on them. It is impossible to achieve good I/O performance with insufficient I/O hardware (for example, too few disks). It is difficult to say how much I/O hardware is sufficient—it depends on the application's requirements, system architecture, performance of the I/O hardware, etc. The vendor of the machine may be able to provide guidance in this regard. Some useful guidelines on how to configure an I/O subsystem are provided in [15].

- **Use Fast File Systems, Not NFS**: On many installations of high-performance machines, the home directories of users are NFS (Network File System [52]) mounted so that they can be accessed directly from other machines. This is a good convenience feature, but users must not use the same directory for reading or writing large amounts of data from

parallel applications, because NFS is terribly slow. They must use the directory that corresponds to the native high-performance file system on the machine.

- **Do Not Perform I/O From One Process Only**: Many parallel applications still perform I/O by having all processes send their data to one process that gathers all the data and writes it to a file. Application developers have chosen this approach because of historical limitations in the I/O capabilities of many parallel systems: either parallel I/O from multiple processes to a common file was not supported, or, if supported, the performance was poor. On modern parallel systems, however, these limitations no longer exist. With sufficient and appropriately configured I/O hardware and modern high-performance file systems, one can achieve higher performance by having multiple processes directly access a common file. The MPI-IO interface is specifically designed to support such accesses and to enable implementations to deliver high performance for such accesses.

- **Make Large Requests Wherever Possible**: I/O performance is much higher for large requests than for small requests. Application developers must therefore make an attempt to write their programs in a way that they make large I/O requests rather than lots of small requests, wherever possible.

- **Use MPI-IO and Use it the Right Way**: MPI-IO offers great potential in terms of portability and high performance. It gives implementations an opportunity to optimize I/O. Therefore, we recommend that users use MPI-IO and use it the right way. The right way is explained in more detail below, but, in short, whenever each process needs to access noncontiguous data and multiple processes need to perform such I/O, users must use MPI derived datatypes, define a file view, and use a single collective I/O function. They must not access each small contiguous piece separately as they would with Unix I/O.

13.5.2 Achieving High Performance with MPI-IO

Let us examine the different ways of writing an application with MPI-IO and see how this choice affects performance.

Any application has a particular "I/O access pattern" based on its I/O needs. The same I/O access pattern, however, can be presented to the I/O system in different ways, depending on which I/O functions the application uses and how. The different ways of expressing I/O access patterns in MPI-IO can be classified into four *levels*, level 0 through level 3 [57]. We explain this classification with the help of the same example we considered in previous sections, namely, accessing a distributed array from a file (Figure 13.3). The principle applies to other access patterns as well.

Recall that in this example the local array of each process is not contiguous in the file; each row of the local array is separated by rows of the local arrays of

```
MPI_File_open(..., "filename", ..., &fh)
for (i=0; i<n_local_rows; i++) {
   MPI_File_seek(fh, ...)
   MPI_File_read(fh, row[i], ...)
}
MPI_File_close(&fh)
```

Level 0
(many independent, contiguous requests)

```
MPI_File_open(MPI_COMM_WORLD, "filename", ..., &fh)
for (i=0; i<n_local_rows; i++) {
   MPI_File_seek(fh, ...)
   MPI_File_read_all(fh, row[i], ...)
}
MPI_File_close(&fh)
```

Level 1
(many collective, contiguous requests)

```
MPI_Type_create_subarray(..., &subarray, ...)
MPI_Type_commit(&subarray)
MPI_File_open(..., "filename", ..., &fh)
MPI_File_set_view(fh, ..., subarray, ...)
MPI_File_read(fh, local_array, ...)
MPI_File_close(&fh)
```

Level 2
(single independent, noncontiguous request)

```
MPI_Type_create_subarray(.., &subarray, ...)
MPI_Type_commit(&subarray)
MPI_File_open(MPI_COMM_WORLD, "filename", ..., &fh)
MPI_File_set_view(fh, ..., subarray, ...)
MPI_File_read_all(fh, local_array, ...)
MPI_File_close(&fh)
```

Level 3
(single collective, noncontiguous request)

Figure 13.8: Pseudo-code that shows four ways of accessing the data in Figure 13.3 with MPI-IO

other processes. Figure 13.8 shows four ways in which a user can express this access pattern in MPI-IO. In level 0, each process does Unix-style accesses—one independent read request for each row in the local array. Level 1 is similar to level 0 except that it uses collective I/O functions, which indicate to the implementation that all processes that together opened the file will call this function, each with its own access information. Independent I/O functions, on the other hand, convey no information about what other processes will do. In level 2, each process creates a derived datatype to describe the noncontiguous access pattern, defines a file view, and calls independent I/O functions. Level 3 is similar to level 2 except that it uses collective I/O functions.

The four levels represent increasing amounts of data per request, as illustrated in Figure 13.9.[4] The more the amount of data per request, the greater is the opportunity for the implementation to deliver higher performance. How good the performance is at each level depends, of course, on how well the implementation takes advantage of the extra access information at each level. ROMIO, for example, performs data sieving for level-2 requests and collective I/O for level-3 requests. However, it cannot perform these optimizations if the user does not express the access pattern in terms of level-2 or level-3 requests. Users must therefore strive to express their I/O requests as level 3 rather than level 0. Figure 13.10 shows the detailed code for creating a derived datatype,

[4]In this figure, levels 1 and 2 represent the same amount of data per request, but, in general, when the number of noncontiguous accesses per process is greater than the number of processes, level 2 represents more data than level 1.

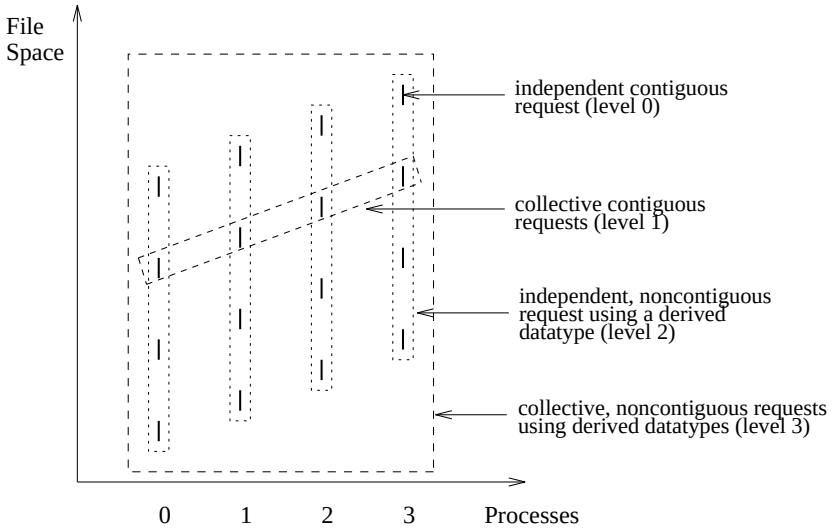

Figure 13.9: The four levels representing increasing amounts of data per request

defining a file view, and making a level-3 I/O request for the distributed-array example of Figure 13.3.

If an application needs to access only large, contiguous pieces of data, level 0 is equivalent to level 2, and level 1 is equivalent to level 3. Users need not create derived datatypes in such cases, as level-0 requests themselves will likely perform well. Many real parallel applications, however, do not fall into this category [3, 12, 36, 50, 51, 56].

We note that the MPI standard does not *require* an implementation to perform any of these optimizations. Nevertheless, even if an implementation does not perform any optimization and instead translates level-3 requests into several level-0 requests to the file system, the performance would be no worse than if the user directly made level-0 requests. Therefore, there is no reason not to use level-3 requests (or level-2 requests where level-3 requests are not possible).

Performance Results

We present some performance results to demonstrate how the choice of level of request affects performance. We wrote the distributed-array access example using level-0, level-2, and level-3 requests and ran the three versions *portably* on five different parallel machines—HP Exemplar, SGI Origin2000, IBM SP, Intel Paragon, and NEC SX-4—using ROMIO. (For this particular application, level-1 requests do not contain sufficient information for any useful optimizations, and ROMIO therefore internally translates level-1 requests into level-0 requests.) We used the native file systems on each machine: HFS on the Exemplar, XFS on the Origin2000, PIOFS on the SP, PFS on the Paragon, and SFS on the SX-4.

We note that the machines had varying amounts of I/O hardware. Some

```
gsizes[0] = num_global_rows;
gsizes[1] = num_global_cols;
distribs[0] = distribs[1] = MPI_DISTRIBUTE_BLOCK;
dargs[0] = dargs[1] = MPI_DISTRIBUTE_DFLT_DARG;
psizes[0] = psizes[1] = 4;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Type_create_darray(16, rank, 2, gsizes, distribs, dargs,
                       psizes, MPI_ORDER_C, MPI_FLOAT,
                       &filetype);
MPI_Type_commit(&filetype);
local_array_size = num_local_rows * num_local_cols;
MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile", MPI_MODE_RDONLY,
              MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, 0, MPI_FLOAT, filetype, "native",
                  MPI_INFO_NULL);
MPI_File_read_all(fh, local_array, local_array_size,
                  MPI_FLOAT, &status);
MPI_File_close(&fh);
```

Figure 13.10: Detailed code for the distributed-array example of Figure 13.3 using a level-3 request

of the differences in performance results among the machines are due to these variations. Our goal in this experiment was to compare the performance of the different levels of requests on a given machine, rather than comparing the performance of different machines.

Figures 13.11 and 13.12 show the read and write bandwidths. The performance with level-0 requests was, in general, very poor because level-0 requests result in too many small read/write calls. For level-2 requests—for which ROMIO performs data sieving—the read bandwidth improved over level-0 requests by a factor ranging from 2.6 on the HP Exemplar to 453 on the NEC SX-4. Similarly, the write bandwidth improved by a factor ranging from 2.3 on the HP Exemplar to 121 on the NEC SX-4. The performance improved considerably with level-3 requests because ROMIO performs collective I/O in this case. The read bandwidth improved by a factor of as much as 793 over level-0 requests (NEC SX-4) and as much as 14 over level-2 requests (Intel Paragon). Similarly, with level-3 requests, the write performance improved by a factor of as much as 721 over level-0 requests (NEC SX-4) and as much as 40 over level-2 requests (HP Exemplar). It is clearly advantageous to use level-3 requests rather than any other kind of request.

We obtained similar results with other applications as well; see [58] for details.

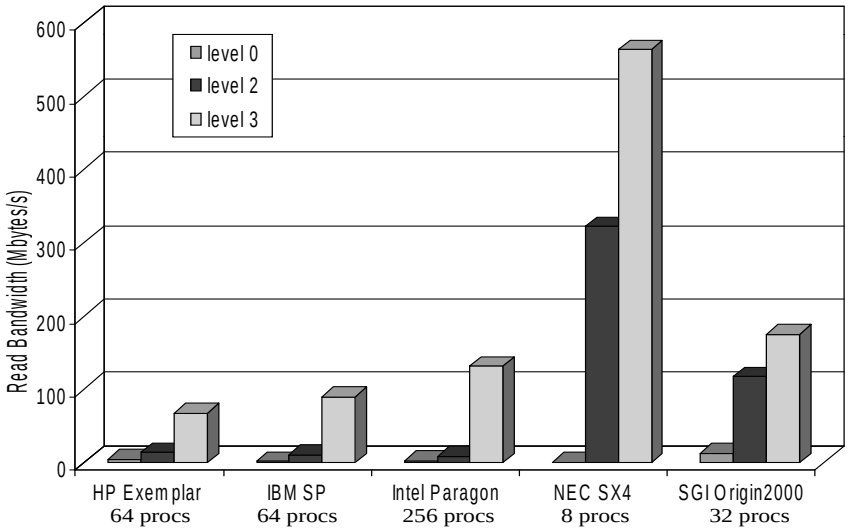

Figure 13.11: Read performance of distributed array access (array size 512x512x512 integers = 512 Mbytes)

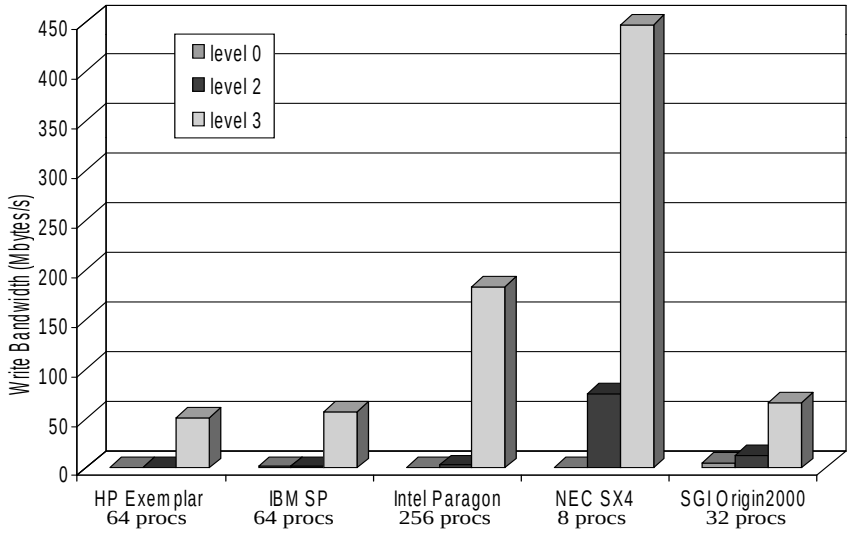

Figure 13.12: Write performance of distributed array access (array size 512x512x512 integers = 512 Mbytes)

Upshot Graphs

We present some graphs that illustrate the reduction in time obtained by using level-2 and level-3 requests instead of level-0 requests for writing a three-dimensional distributed array of size $128 \times 128 \times 128$ on 32 processors on the Intel Paragon at Caltech. We instrumented the ROMIO source code to measure the time taken for each file-system call made by ROMIO and also for the computation and communication required for collective I/O. The instrumented code created trace files, which we visualized using a performance-visualization tool called *Upshot* [22].

Figure 13.13 shows the Upshot plot for level-0 requests, where each process makes a separate write function call to write each row of its local array. The numerous small bands represent the numerous writes in the program, as a result of which the total time taken is about 125 seconds. The large white portions are actually lots of writes clustered together, which become visible when you zoom in to the region using Upshot.

Figure 13.14 shows the Upshot plot for level-2 requests, for which ROMIO performs performs data sieving. In this case it performed data sieving in blocks of 4 Mbytes at a time. Notice that the total time has decreased to about 16 seconds compared with 125 seconds for level-0 requests. For writing with data sieving, each process must perform a read-modify-write and also lock the region of the file being written. Because of the need for file locking and a buffer size of 4 Mbytes, many processes remain idle waiting to acquire locks. Therefore, only a few write operations take place concurrently. It should be possible to increase parallelism, however, by decreasing the size of the buffer used for data sieving. Figure 13.15 shows the results for a buffer size of 512 Kbytes. Since more I/O operations take place in parallel, the total time decreased to 10.5 seconds. A further reduction in buffer size to 64 Kbytes (Figure 13.16) resulted in even greater parallelism, but the I/O time increased because of the smaller granularity of each I/O operation. The performance of data sieving can thus be tuned by varying the size of the buffer used for data sieving, which can be done via the hints mechanism in MPI-IO.

Figure 13.17 shows the Upshot plot for level-3 requests, for which ROMIO performs collective I/O. The total time decreased to about 2.75 seconds, which means that level-3 requests were about 45 times faster than level-0 requests and about four times faster than the best performance with level-2 requests. The reason for the improvement is that the numerous writes of each process were coalesced into a *single* write at the expense of some extra computation (to figure out how to merge the requests) and interprocess communication. With collective I/O, the actual write time was only a small fraction of the total I/O time; for example, file open took longer than the write.

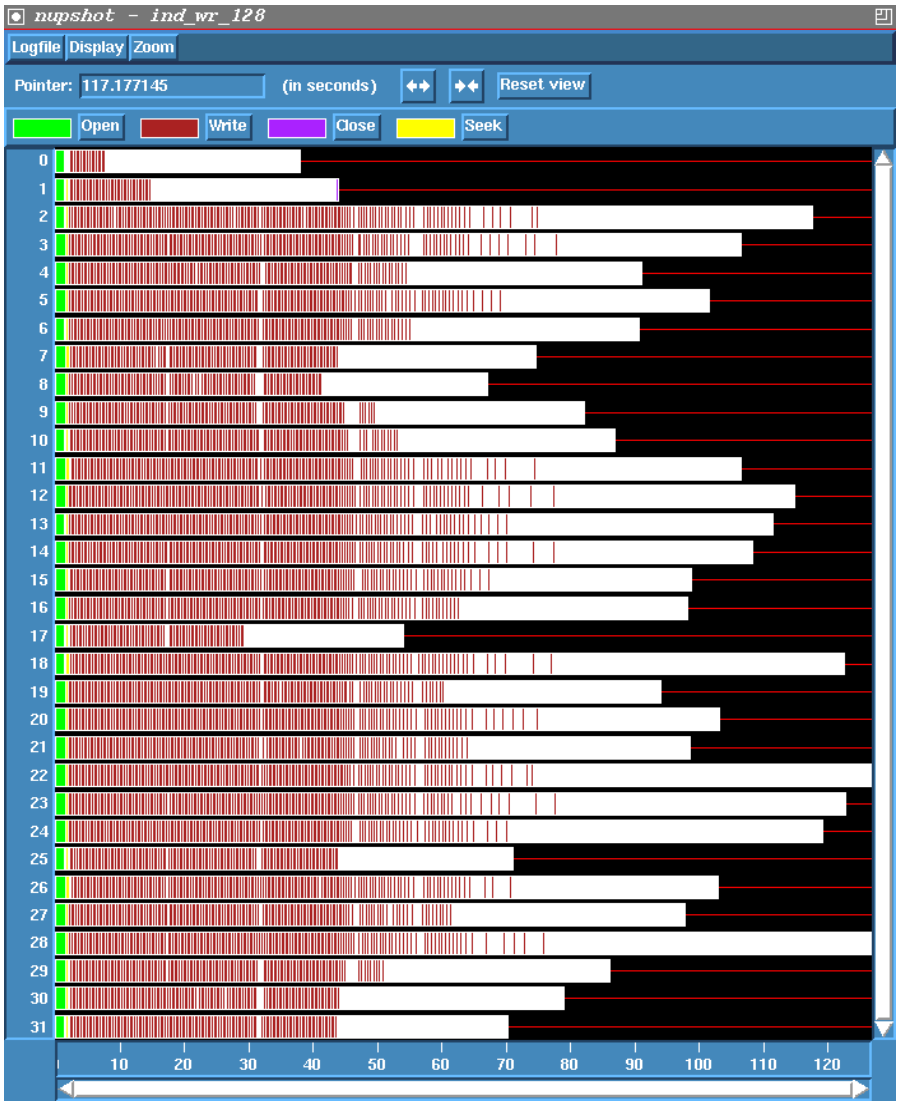

Figure 13.13: Writing a $128 \times 128 \times 128$ distributed array on the Intel Paragon using level-0 requests (Unix-style independent writes). Elapsed time = 125 seconds.

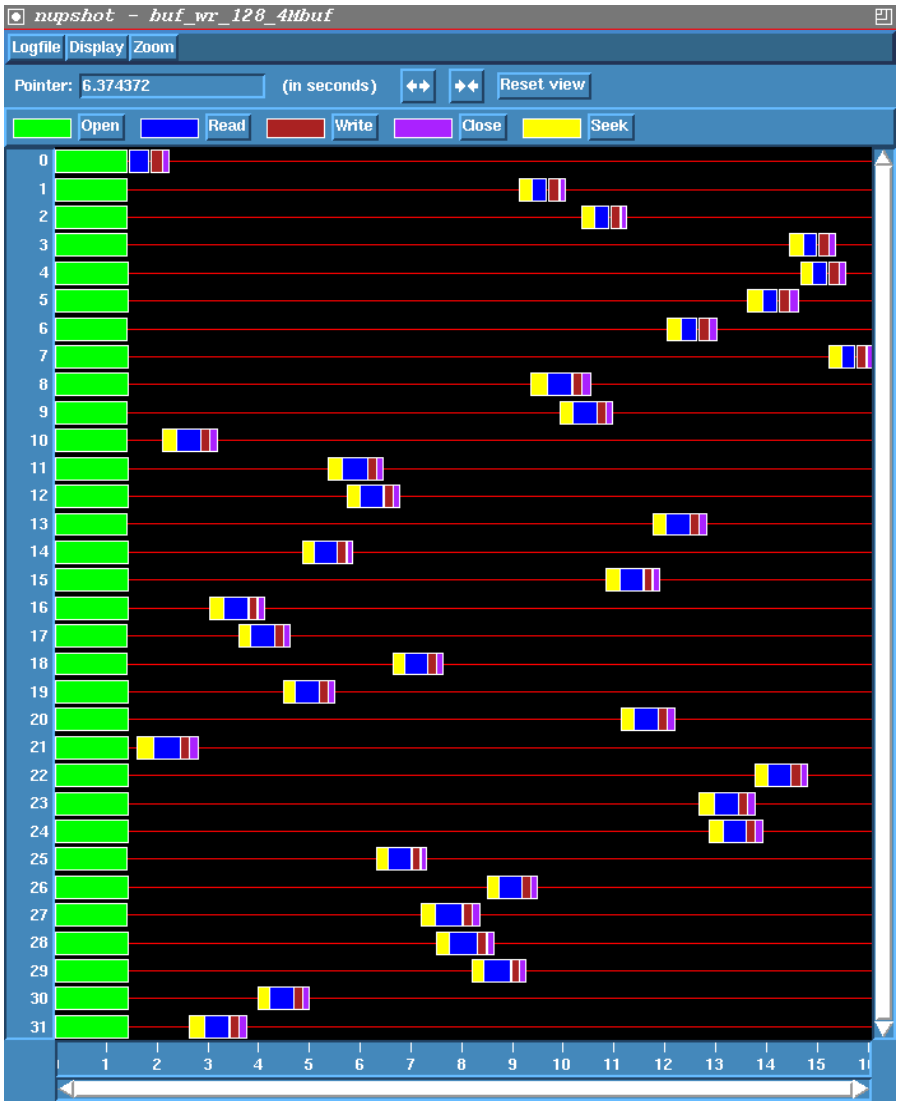

Figure 13.14: Writing a $128 \times 128 \times 128$ distributed array on the Intel Paragon using level-2 requests, with the buffer size for data sieving = 4 Mbytes. Elapsed time = 16 seconds.

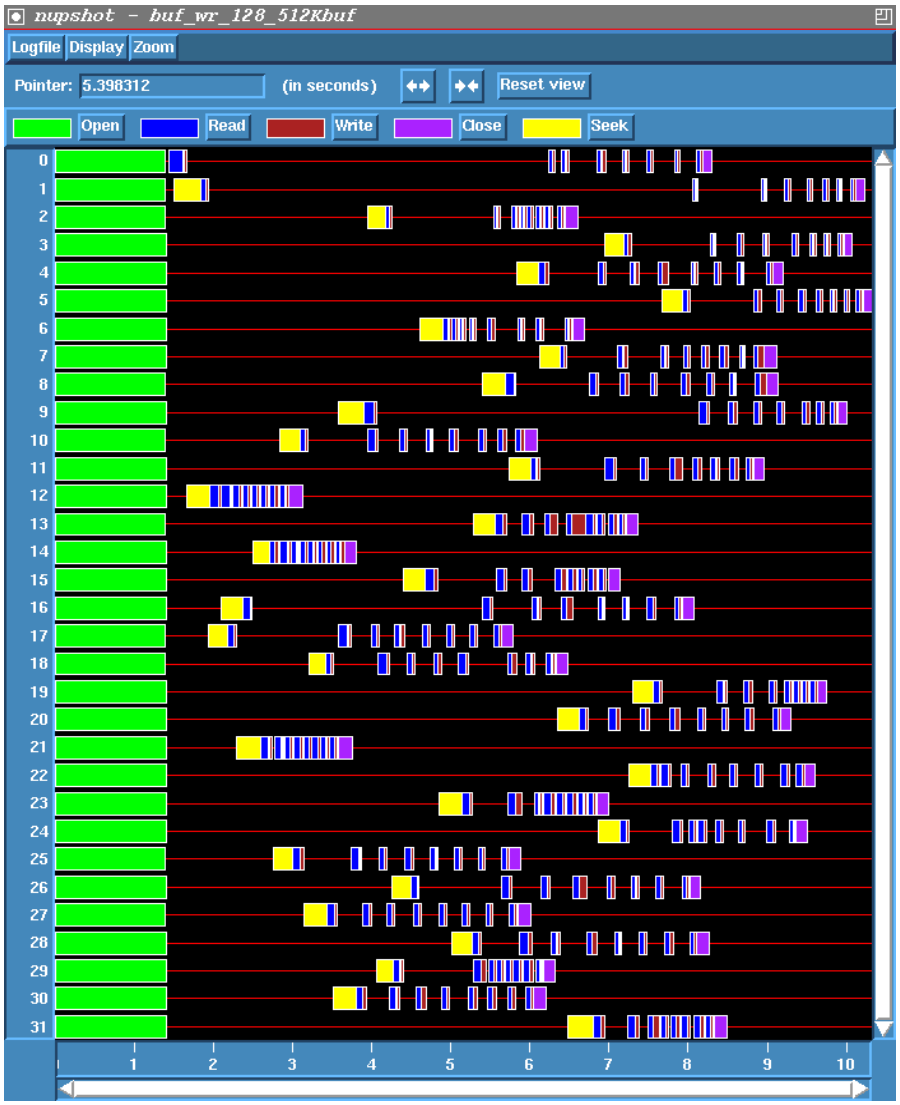

Figure 13.15: Writing a $128 \times 128 \times 128$ distributed array on the Intel Paragon using level-2 requests, with the buffer size for data sieving = 512 Kbytes. Elapsed time = 10.5 seconds.

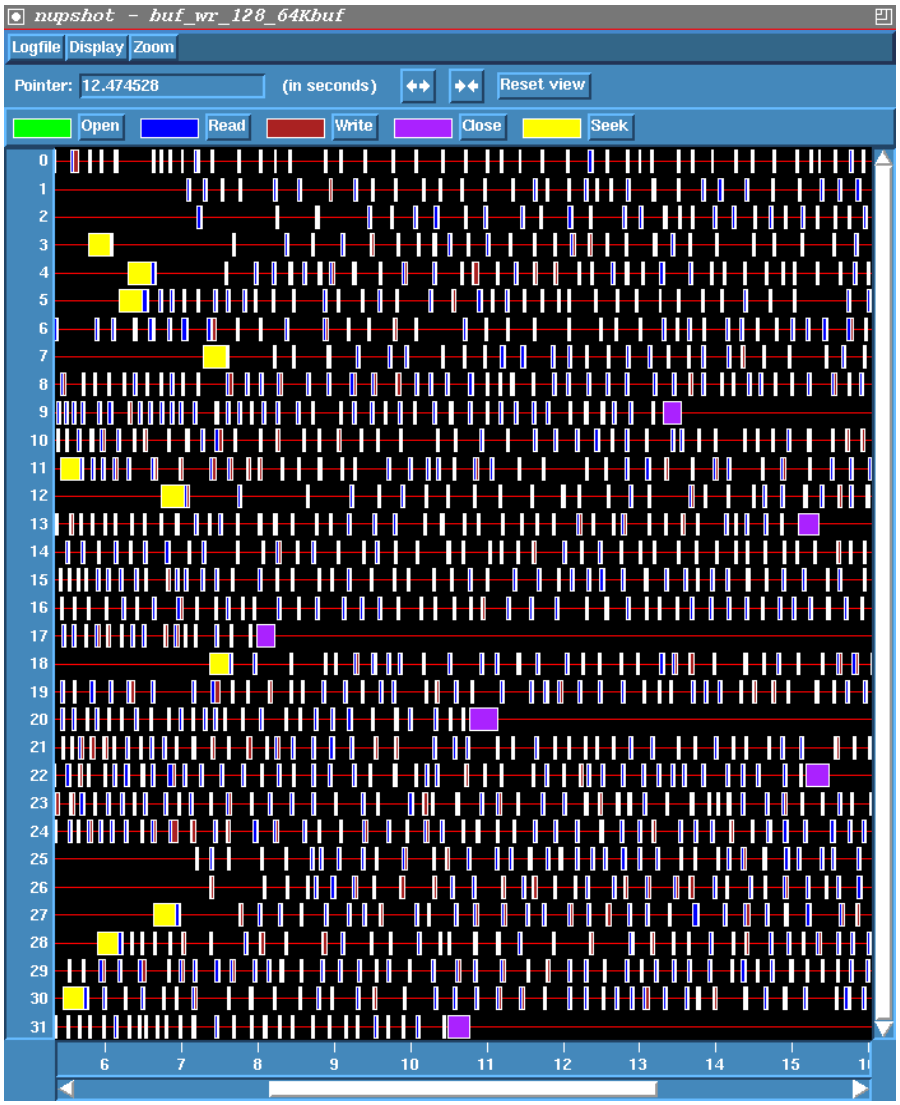

Figure 13.16: Writing a $128 \times 128 \times 128$ distributed array on the Intel Paragon using level-2 requests, with the buffer size for data sieving = 64 Kbytes. Elapsed time = 20 seconds.

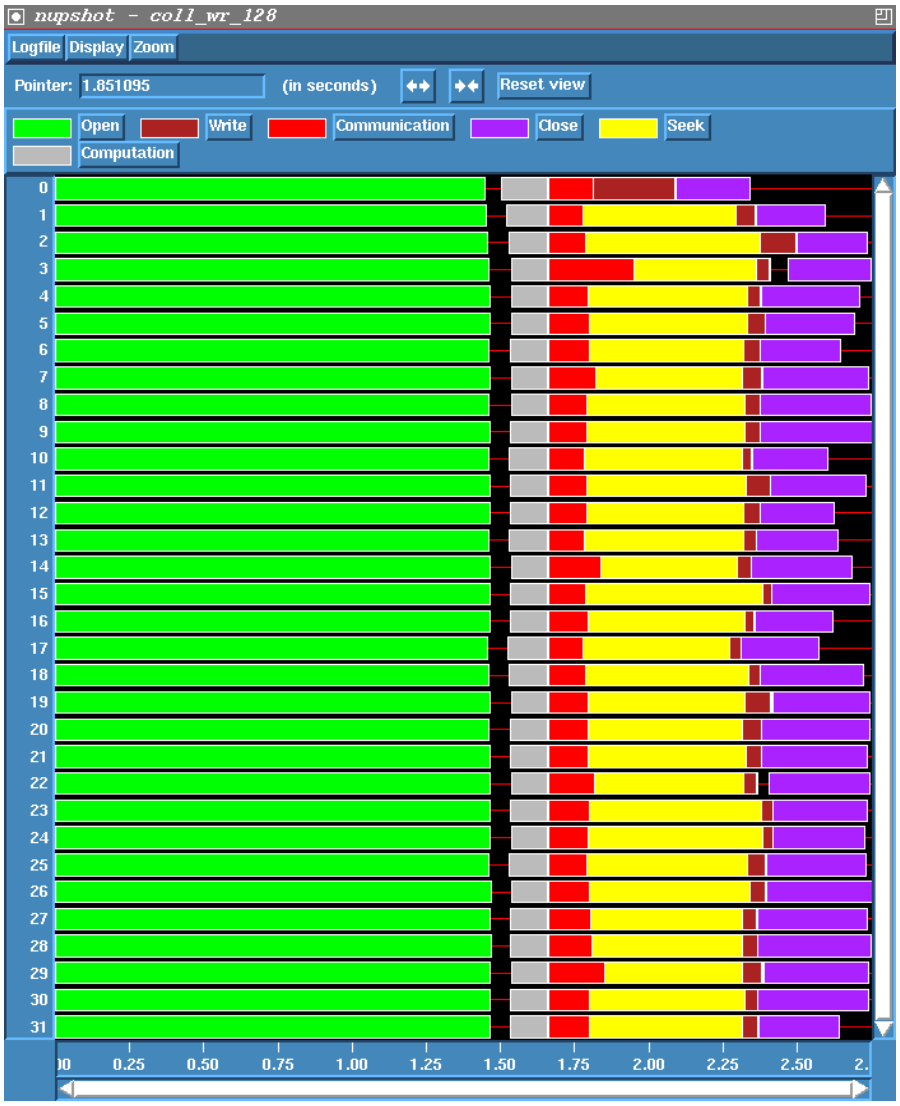

Figure 13.17: Writing a $128 \times 128 \times 128$ distributed array on the Intel Paragon using level-3 requests. Elapsed time = 2.75 seconds.

13.6 Summary

I/O on parallel computers has always been slow compared with computation and communication. As computers get larger and faster, I/O becomes even more of a problem. In this chapter we have provided a general introduction to the field of parallel I/O. Our emphasis has been on the practical aspects of *using* parallel I/O and achieving high performance. By following the guidelines presented, we believe that users *can* achieve high I/O performance in parallel applications.

Acknowledgments

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

Bibliography

[1] An introduction to GPFS 1.2. `http://www.rs6000.ibm.com/resource/technology/paper1.html`.

[2] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 109–126. ACM Press, December 1995.

[3] Sandra Johnson Baylor and C. Eric Wu. Parallel I/O workload characteristics using Vesta. In R. Jain, J. Werth, and J. Browne, editors, *Input/Output in Parallel and Distributed Computer Systems*, chapter 7, pages 167–185. Kluwer Academic Publishers, 1996.

[4] Rajesh Bordawekar. *Techniques for Compiling I/O Intensive Parallel Programs*. PhD thesis, Electrical and Computer Engineering Dept., Syracuse University, April 1996. Also available as Caltech technical report CACR-118.

[5] Rajesh Bordawekar, Alok Choudhary, Ken Kennedy, Charles Koelbel, and Michael Paleczny. A model and compilation strategy for out-of-core data parallel programs. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 1–10. ACM Press, July 1995.

[6] Pei Cao, Edward Felten, Anna Karlin, and Kai Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, November 1996.

[7] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[8] Coda file system. `http://www.coda.cs.cmu.edu`.

[9] Peter Corbett, Dror Feitelson, Yarsun Hsu, Jean-Pierre Prost, Marc Snir, Sam Fineberg, Bill Nitzberg, Bernard Traversat, and Parkson Wong. MPI-IO: a parallel file I/O interface for MPI. Technical Report IBM Research

Report RC 19841(87784), IBM T.J. Watson Research Center, November 1994.

[10] Peter F. Corbett and Dror G. Feitelson. The Vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3):225–264, August 1996.

[11] Peter F. Corbett, Jean-Pierre Prost, Chris Demetriou, Garth Gibson, Erik Reidel, Jim Zelenka, Yuqun Chen, Ed Felten, Kai Li, John Hartman, Larry Peterson, Brian Bershad, Alec Wolman, and Ruth Aydt. Proposal for a common parallel file system programming interface. WWW http://www.cs.arizona.edu/sio/api1.0.ps, September 1996. Version 1.0.

[12] Phyllis E. Crandall, Ruth A. Aydt, Andrew A. Chien, and Daniel A. Reed. Input-output characteristics of scalable parallel applications. In *Proceedings of Supercomputing '95*. ACM Press, December 1995.

[13] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *Proceedings of the Workshop on I/O in Parallel Computer Systems at IPPS '93*, pages 56–70, April 1993. Also published in *Computer Architecture News*, 21(5):31–38, December 1993.

[14] Juan Miguel del Rosario and Alok Choudhary. High performance I/O for parallel computers: Problems and prospects. *Computer*, 27(3):59–68, March 1994.

[15] Dror G. Feitelson, Peter F. Corbett, Sandra Johnson Baylor, and Yarsun Hsu. Parallel I/O subsystems in massively parallel supercomputers. *IEEE Parallel and Distributed Technology*, 3(3):33–47, Fall 1995.

[16] Fibre channel industry association. `http://www.fibrechannel.com`.

[17] Samuel A. Fineberg, Parkson Wong, Bill Nitzberg, and Chris Kuszmaul. PMPIO—a portable implementation of MPI-IO. In *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 188–195. IEEE Computer Society Press, October 1996.

[18] N. Galbreath, W. Gropp, and D. Levine. Applications-driven parallel I/O. In *Proceedings of Supercomputing '93*, pages 462–471. IEEE Computer Society Press, November 1993.

[19] Garth A. Gibson, David P. Nagle, Khalil Amiri, Fay W. Chang, Eugene Feinberg, Howard Gobioff Chen Lee, Berend Ozceri, Erik Riedel, and David Rochberg. A case for network-attached secure disks. Technical Report CMU–CS-96-142, Carnegie-Mellon University, June 1996.

[20] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI—The Complete Reference: Volume 2, The MPI-2 Extensions*. MIT Press, Cambridge, MA, 1998.

[21] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.

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

[23] Jay Huber, Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal. PPFS: A high performance portable parallel file system. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 385–394. ACM Press, July 1995.

[24] IEEE/ANSI Std. 1003.1. Portable operating system interface (POSIX)–part 1: System application program interface (API) [C language], 1996 edition.

[25] Terry Jones, Richard Mark, Jeanne Martin, John May, Elsie Pierce, and Linda Stanberry. An MPI-IO interface to HPSS. In *Proceedings of the Fifth NASA Goddard Conference on Mass Storage Systems*, pages I:37–50, September 1996. Also available from `http://esdis-it.gsfc.nasa.gov/MSST/conf1998.html`.

[26] David Kotz. Applications of parallel I/O. Technical Report PCS-TR96-297, Dept. of Computer Science, Dartmouth College, October 1996. Release 1. `http://www.cs.dartmouth.edu/reports/abstracts/TR96-297`.

[27] David Kotz. Introduction to multiprocessor I/O architecture. In Ravi Jain, John Werth, and James C. Browne, editors, *Input/Output in Parallel and Distributed Computer Systems*, chapter 4, pages 97–123. Kluwer Academic Publishers, 1996.

[28] David Kotz. Disk-directed I/O for MIMD multiprocessors. *ACM Transactions on Computer Systems*, 15(1):41–74, February 1997.

[29] Tara M. Madhyastha and Daniel A. Reed. Intelligent, adaptive file system policy selection. In *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 172–179. IEEE Computer Society Press, October 1996.

[30] Tara M. Madhyastha and Daniel A. Reed. Exploiting global input/output access pattern classification. In *Proceedings of SC97: High Performance Networking and Computing*. ACM Press, November 1997.

[31] Message Passing Interface Forum. MPI-2: Extensions to the message-passing interface, July 1997. `http://www.mpi-forum.org/docs/docs.html`.

[32] Message Passing Interface Forum. MPI: A message-passing interface standard, version 1.1, June 1995. `http://www.mpi-forum.org/docs/docs.html`.

[33] NetCDF. `http://www.unidata.ucar.edu/packages/netcdf`.

[34] Jarek Nieplocha, Ian Foster, and Rick Kendall. ChemIO: High-performance parallel I/O for computational chemistry applications. *The International Journal of High Performance Computing Applications*, 12(3):345–363, Fall 1998.

[35] Nils Nieuwejaar and David Kotz. The Galley parallel file system. *Parallel Computing*, 23(4):447–476, June 1997.

[36] Nils Nieuwejaar, David Kotz, Apratim Purakayastha, Carla Schlatter Ellis, and Michael Best. File-access characteristics of parallel scientific workloads. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1075–1089, October 1996.

[37] Bill Nitzberg and Virginia Lo. Collective buffering: Improving parallel I/O performance. In *Proceedings of the Sixth IEEE International Symposium on High Performance Distributed Computing*, pages 148–157. IEEE Computer Society Press, August 1997.

[38] Applications Working Group of the Scalable I/O Initiative. Preliminary survey of I/O intensive applications. Scalable I/O Initiative Working Paper Number 1. `http://www.cacr.caltech.edu/SIO/pubs/SIO_apps.ps`, 1994.

[39] Michael Paleczny, Ken Kennedy, and Charles Koelbel. Compiler support for out-of-core arrays on data parallel machines. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 110–118, February 1995.

[40] David Patterson, Garth Gibson, and Randy Katz. A case for redundant arrays of inexpensive disks. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 109–116. ACM Press, June 1988.

[41] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *Proceedings of the 15th Symposium on Operating System Principles*, pages 79–95. ACM Press, December 1995.

[42] Paul Pierce. A concurrent file system for a highly parallel mass storage subsystem. In *Proceedings of* 4^{th} *Conference on Hypercubes, Concurrent Computers and Applications*, pages 155–160. Golden Gate Enterprises, Los Altos, CA, March 1989.

[43] PMPIO - a portable MPI-2 I/O library. `http://parallel.nas.nasa.gov/MPI-IO/pmpio/pmpio.html`.

[44] Jean-Pierre Prost, Marc Snir, Peter Corbett, and Dror Feitelson. MPI-IO, a message-passing interface for concurrent I/O. Technical Report RC 19712 (87394), IBM T.J. Watson Research Center, August 1994.

[45] ROMIO: A high-performance, portable MPI-IO implementation. `http://www.mcs.anl.gov/romio`.

[46] Scalable I/O initiative. `http://www.cacr.caltech.edu/SIO`.

[47] Scientific data management. `http://www.ca.sandia.gov/asci-sdm`.

[48] K. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95*. ACM Press, December 1995.

[49] Huseyin Simitci and Daniel Reed. A comparison of logical and physical parallel I/O patterns. *The International Journal of High Performance Computing Applications*, 12(3):364–380, Fall 1998.

[50] E. Smirni and D. A. Reed. Lessons from characterizing the input/output behavior of parallel scientific applications. *Performance Evaluation: An International Journal*, 33(1):27–44, June 1998.

[51] Evgenia Smirni, Ruth A. Aydt, Andrew A. Chien, and Daniel A. Reed. I/O requirements of scientific applications: An evolutionary view. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, pages 49–59. IEEE Computer Society Press, 1996.

[52] Hal Stern. *Managing NFS and NIS*. O'Reilly & Associates, Inc., 1991.

[53] Rajeev Thakur, Rajesh Bordawekar, Alok Choudhary, Ravi Ponnusamy, and Tarvinder Singh. PASSION runtime library for parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 119–128. IEEE Computer Society Press, October 1994.

[54] Rajeev Thakur and Alok Choudhary. An extended two-phase method for accessing sections of out-of-core arrays. *Scientific Programming*, 5(4):301–317, Winter 1996.

[55] Rajeev Thakur, Alok Choudhary, Rajesh Bordawekar, Sachin More, and Sivaramakrishna Kuditipudi. Passion: Optimized I/O for parallel applications. *Computer*, 29(6):70–78, June 1996.

[56] Rajeev Thakur, William Gropp, and Ewing Lusk. An experimental evaluation of the parallel I/O systems of the IBM SP and Intel Paragon using a production application. In *Proceedings of the 3rd International Conference of the Austrian Center for Parallel Computation (ACPC) with Special Emphasis on Parallel Databases and Parallel I/O*, pages 24–35. Lecture Notes in Computer Science 1127. Springer-Verlag, September 1996.

[57] Rajeev Thakur, William Gropp, and Ewing Lusk. A case for using MPI's derived datatypes to improve I/O performance. In *Proceedings of SC98: High Performance Networking and Computing*, November 1998.

[58] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189. IEEE Computer Society Press, February 1999.

[59] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems*, pages 23–32. ACM Press, May 1999.

[60] The global file system. `http://gfs.lcse.umn.edu`.

[61] The hard disk drive guide. `http://www.storagereview.com/guide`.

[62] The MPI-IO Committee. MPI-IO: A parallel file I/O interface for MPI, version 0.5.

[63] The NCSA HDF home page. `http://hdf.ncsa.uiuc.edu`.

[64] The parallel virtual file system. `http://www.parl.clemson.edu/pvfs`.

[65] Sivan Toledo and Fred G. Gustavson. The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computations. In *Proceedings of Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 28–40. ACM Press, May 1996.

[66] Len Wisniewski, Brad Smisloff, and Nils Nieuwejaar. Sun MPI I/O: Efficient I/O for parallel applications. In *Proceedings of SC99: High Performance Networking and Computing*, November 1999.