Appeared in *Proc. of the 7th Symposium on the Frontiers of Massively Parallel Computation*, February 1999, pp. 182–189. 

# Data Sieving and Collective I/O in ROMIO

Rajeev Thakur William Gropp Ewing Lusk
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, USA
{thakur, gropp, lusk}@mcs.anl.gov

## Abstract

*The I/O access patterns of parallel programs often consist of accesses to a large number of small, noncontiguous pieces of data. If an application's I/O needs are met by making many small, distinct I/O requests, however, the I/O performance degrades drastically. To avoid this problem, MPI-IO allows users to access a noncontiguous data set with a single I/O function call. This feature provides MPI-IO implementations an opportunity to optimize data access.*

*We describe how our MPI-IO implementation, ROMIO, delivers high performance in the presence of noncontiguous requests. We explain in detail the two key optimizations ROMIO performs: data sieving for noncontiguous requests from one process and collective I/O for noncontiguous requests from multiple processes. We describe how one can implement these optimizations portably on multiple machines and file systems, control their memory requirements, and also achieve high performance. We demonstrate the performance and portability with performance results for three applications—an astrophysics-application template (DIST3D), the NAS BTIO benchmark, and an unstructured code (UNSTRUC)—on five different parallel machines: HP Exemplar, IBM SP, Intel Paragon, NEC SX-4, and SGI Origin2000.*

## 1 Introduction

Numerous studies of the I/O characteristics of parallel applications have shown that many applications need to access a large number of small, noncontiguous pieces of data from a file [1, 2, 7, 9, 10]. For good I/O performance, however, the size of an I/O request must be large (on the order of megabytes). The I/O performance suffers considerably if applications access data by making many small I/O requests. Such is the case when parallel applications perform I/O by using the Unix `read` and `write` functions, which can access only a single contiguous chunk of data at a time.

MPI-IO, the I/O part of the MPI-2 standard [6], is a new interface designed specifically for portable, high-performance parallel I/O. To avoid the above-mentioned problem of many distinct, small I/O requests, MPI-IO allows users to specify the entire noncontiguous access pattern and read or write all the data with a single I/O function call. MPI-IO also allows users to specify collectively the I/O requests of a group of processes, thereby providing the implementation with even greater access information and greater scope for optimization.

In this paper we describe how our MPI-IO implementation, ROMIO, delivers high performance in the presence of noncontiguous I/O requests. ROMIO is a portable MPI-IO implementation that works on many different machines and file systems. We explain in detail the two key optimizations ROMIO performs: data sieving for noncontiguous requests from one process and collective I/O for noncontiguous requests from multiple processes. We describe how one can implement these optimizations portably on multiple machines and file systems, control their memory requirements, and also achieve high performance. We demonstrate the performance and portability with performance results for three applications on five different parallel machines.

We note that ROMIO can perform the optimizations described in this paper only if users provide complete access information in a single function call. In [14] we explained how users can do so by using MPI's derived datatypes to create file views and by using MPI-IO's collective-I/O functions whenever possible. In this paper we describe the optimizations in detail and provide extensive performance results.

The rest of this paper is organized as follows. Section 2 gives a brief overview of ROMIO. Data sieving is described in Section 3 and collective I/O in Section 4. Performance results are presented in Section 5, followed by conclusions in Section 6.

## 2 Overview of ROMIO

ROMIO is a freely available, high-performance, portable implementation of MPI-IO. The current version of ROMIO, 1.0.1, runs on the following machines: IBM SP; Intel Paragon; HP Exemplar; SGI Origin2000 and T3E; NEC SX-4; other symmetric multiprocessors from HP, SGI, Sun, DEC, and IBM; and networks of workstations (Sun, SGI, HP, IBM, DEC, Linux, and FreeBSD). Supported file systems are IBM PIOFS, Intel PFS, HP HFS, SGI XFS, NEC SFS, NFS, and any Unix file system (UFS). ROMIO 1.0.1 includes everything defined in the MPI-2 I/O chapter except shared-file-pointer functions, split-collective-I/O functions, support for file interoperability, I/O error handling, and I/O error classes. ROMIO is designed to be used with *any* MPI-1 implementation—both portable and vendor-specific implementations. It is currently included as part of three MPI implementations: MPICH, HP MPI, and SGI MPI.

A key component of ROMIO that enables such a portable MPI-IO implementation is an internal layer called ADIO [13]. ADIO, an abstract-device interface for I/O, consists of a small set of basic functions for parallel I/O. In ROMIO, the MPI-IO interface is implemented portably on top of ADIO, and only ADIO is implemented separately for different file systems. ADIO thus separates the machine-dependent and machine-independent aspects involved in implementing MPI-IO.

## 3 Data Sieving

To reduce the effect of high I/O latency, it is critical to make as few requests to the file system as possible. When a process makes an independent request for noncontiguous data, ROMIO, therefore, does not access each contiguous portion of the data separately. Instead, it uses an optimization called data sieving [12]. The basic idea is illustrated in Figure 1. Assume that the user has made a single read request for five noncontiguous pieces of data. Instead of reading each 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 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 bytes requested (including holes). The extent can potentially be very large if there are large holes between the requested data segments. 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 default value is 4 Mbytes (per process), but the user can change it 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 could be so large that the cost of reading the extra data outweighs the cost of accessing large chunks. The BTIO benchmark (see Section 5), for example, has such an access pattern. 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. ROMIO also uses another user-controllable parameter that defines the maximum amount of contiguous data that a process can write at a time during data sieving. Since writing requires locking the portion of the file being accessed, ROMIO uses a smaller default buffer size for writing (512 Kbytes) in order to reduce contention for locks.

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. (See the performance results in Section 5.)

## 4 Collective I/O

In many parallel applications, although each process may need to access several noncontiguous portions of a

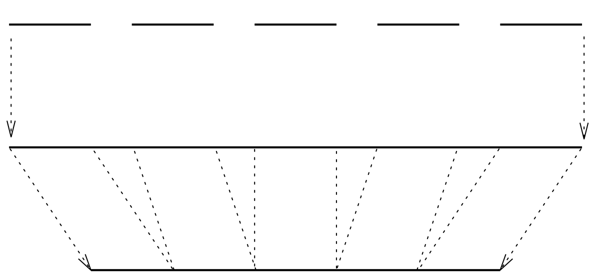


**Figure 1. Data sieving**

file, the requests of different processes are often interleaved and may together span large contiguous portions of the file. If the user provides the MPI-IO implementation with the entire access information of a group of processes, the implementation can improve I/O performance significantly by merging the requests of different processes and servicing the merged request. Such optimization is broadly referred to as collective I/O.

Collective I/O can be performed at the disk level (disk-directed I/O [5]), at the server level (server-directed I/O [8]), or at the client level (two-phase I/O [3]). Since ROMIO is a portable, user-level library with no separate I/O servers, it performs collective I/O at the client level. For this purpose, it uses a generalized version of the extended two-phase method described in [11].

## 4.1 Two-Phase I/O

Two-phase I/O was first proposed in [3] in the context of accessing distributed arrays from files. Consider the example of reading a two-dimensional array from a file into a (block,block) distribution in memory, as shown in Figure 2. Assume that the array is stored in the file in row-major order. As a result of the distribution in memory and the storage order in the file, the local array of each process is located noncontiguously in the file—each row of the local array of a process is separated by rows from the local arrays of other processes. If each process tries to read each row of its local array individually, the performance will be poor due to the large number of relatively small I/O requests. Note, however, that all processes together need to read the entire file, and two-phase I/O uses this fact to improve performance.

If the entire I/O access pattern of all processes is known to the implementation, the data can be accessed efficiently by splitting the access into two phases. In the first phase, processes access data assuming a distribution in memory 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 small compared with the savings in I/O time.

The basic two-phase method was extended in [11] to access sections of out-of-core arrays. In ROMIO we use a generalized version of this extended two-phase method that can handle *any* noncontiguous I/O request as described by an MPI derived datatype, not just sections of arrays.

## 4.2 Generalized Two-Phase I/O in ROMIO

ROMIO uses two user-controllable parameters for collective I/O: the number of processes that perform I/O in the I/O phase and the maximum size on each process of the temporary buffer needed for two-phase I/O. By default, all processes perform I/O in the I/O phase, and the maximum buffer size is 4 Mbytes per process. The user can change these values at run time via MPI-IO's hints mechanism.

We first explain the algorithm ROMIO uses for collective reads and then describe how the algorithm differs for collective writes. Figure 3 shows a simple example that illustrates how ROMIO performs a collective read. In this example, all processes perform I/O, and each process is assumed to have as much memory as needed for the temporary buffer.

We note that, in MPI-IO, the collective-I/O function called by a process specifies the access information of that process only. Also, file accesses in collective I/O refer to accesses from multiple processes to a *common* file.

### 4.2.1 Collective Reads

In ROMIO's implementation of collective reads, each process first analyzes its own I/O request and creates a list of offsets and a list of lengths, where `length[i]` gives the number of bytes that the process needs from location `offset[i]` in the file. Each process also calculates the locations of the first byte (start offset) and the last byte (end offset) it needs from the file and then broadcasts these two

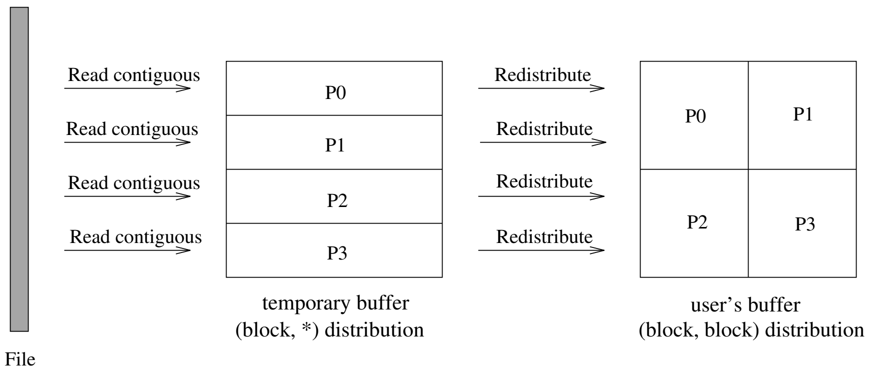


**Figure 2. Reading a distributed array by using two-phase I/O**

offsets to other processes. As a result, each process has the start and end offsets of all processes.

In the next step, each process tries to determine whether this particular access pattern can benefit from collective I/O, i.e., whether the accesses of any of the processes are interleaved in the file. Since an exhaustive check can be expensive, each process only checks if, for any two processes with consecutive ranks ($i$ and $i + 1$), the following expression is true: (start-offset$_{i+1}$ $<$ end-offset$_i$). If not true, each process concludes that collective I/O will not improve performance for this particular access pattern, since the requests of different processes cannot be merged. In such cases, each process just calls the corresponding independent-I/O function, which uses data sieving to optimize noncontiguous requests.

If the above expression is true, the processes proceed to perform collective I/O as follows. Portions of the file are "assigned" to each process such that in the I/O phase of the two-phase operation, a process will access data only from the portion of the file assigned to it. This portion of the file assigned to a process is called the process's *file domain*. If a process needs data located in another process's file domain, it will receive the data from the other process during the communication phase of the two-phase operation. Similarly, if this process's file domain contains data needed by other processes, it must send this data to those processes during the communication phase.

File domains are assigned as follows. Each process calculates the minimum of the start offsets and the maximum of the end offsets of all processes. The difference between these two offsets gives the total extent of the combined request of all processes. The file domain of each process is obtained by dividing this extent equally among the processes.

After the file domains are determined, each process calculates in which other process's file domain its own I/O request (or a portion of it) is located. For each such process, it creates a data structure containing a list of offsets and lengths that specify the data needed from the file domain of that process. It then sends this access information to the processes from which it expects to receive data. Similarly, other processes that need data from the file domain of this process send the corresponding access information to this process. After this exchange has taken place, each process knows what portions of its file domain are needed by other processes and by itself. It also knows which other processes are going to send the data that it needs.

The next step is to read and communicate the data. This step is performed in several parts to reduce its memory requirement. Each process first calculates the offsets corresponding to the first and last bytes needed (by any process) from its file domain. It then divides the difference between these offsets by the maximum size allowed for the temporary buffer (4 Mbytes by default). The result is the number of times (`ntimes`) it needs to perform I/O. All processes then perform a global-maximum operation on `ntimes` to determine the maximum number of times (`max_ntimes`) any process needs to perform I/O. Even if a process has completed all the I/O needed from its own file domain, it may need to participate in communication operations thereafter to receive data from other processes. Each process must therefore be ready to participate in the communication phase `max_ntimes` number of times.

For each of the `ntimes` I/O operations, a process does the following operations: It checks if the current portion of its file domain (no larger than the maximum buffer size) has data that any process needs, including itself. If it does not have such data, the process does not need to perform I/O in this step; it then checks if it needs to receive data from other processes, as explained below. If it does have such data, it reads with a single I/O function call all the data from the first offset to the last offset needed from this portion of the file domain into a temporary buffer in memory. The process effectively performs data sieving, as the data read may include some unwanted data. Now the process must send portions of the data read to processes that need them.

Each process first informs other processes how much

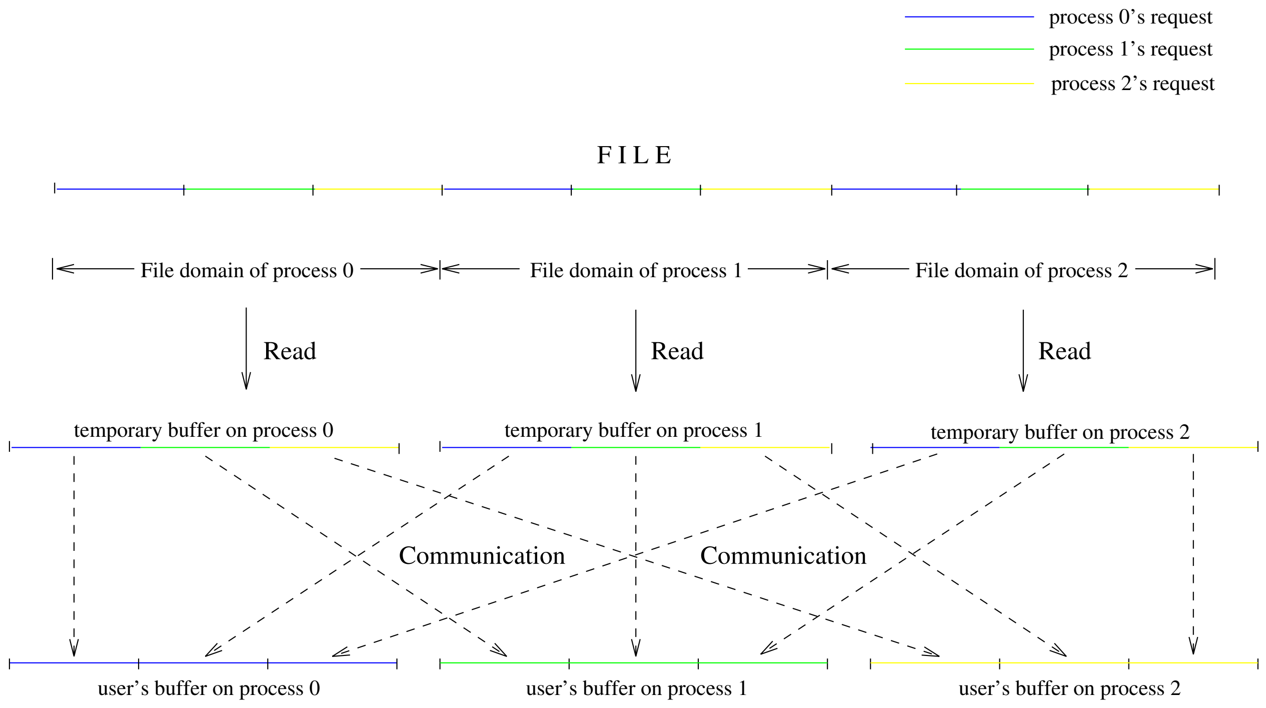


**Figure 3. A simple example illustrating how ROMIO performs a collective read**

data it is going to send them. The processes then exchange data by first posting all the receives as nonblocking operations, then posting all the nonblocking sends, and finally waiting for all the nonblocking communication to complete. MPI derived datatypes are used to send noncontiguous data directly from the temporary buffer to the destination process. On the receive side, if the user has asked for data to be placed contiguously in the user-supplied buffer, the data is received directly into the user's buffer. If data is to be placed noncontiguously, the process first receives data into a temporary buffer and then copies it into the user's buffer. (Since data is received in parts over multiple communication operations from different processes, we found this approach easier than creating derived datatypes on the receive side.)

Each process performs I/O and communication `ntimes` number of times and then participates only in the communication phase for the remaining `(max_ntimes - ntimes)` number of times. In some of these remaining communication steps, a process may not receive any data; nevertheless, the process must check if it is going to receive data in a particular step.

### 4.2.2 Collective Writes

The algorithm for collective writes is similar to the one for collective reads, except that the first phase of the two-phase operation is communication and the second phase is I/O. In the I/O phase, each process checks if any holes (gaps) exist in the data it needs to write. If holes exist, it performs a read-modify-write; otherwise it performs only a write. During the read-modify-write, a process need not lock the region of the file being accessed (unlike in independent I/O), because the process is assured that no other process involved in the collective-I/O operation will directly try to access the data located in this process's file domain. The process is also assured that concurrent writes from processes other than those involved in this collective-I/O operation will not occur, because MPI-IO's consistency semantics [6] do not automatically guarantee consistency for such writes. (In such cases, users must use `MPI_File_sync` and ensure that the operations are not concurrent.)

### 4.2.3 Performance Issues

Even if I/O is performed in large contiguous chunks, the performance of the collective-I/O implementation can be significantly affected by the amount of buffer copying and communication. We were able to improve ROMIO's collective-I/O performance by as much as 50% by tuning the implementation to minimize buffer copying and minimize the number of communication calls and use the right set of MPI communication primitives.

Initially, in each of the communication steps, we always received data into a temporary buffer and then copied it into the user's buffer. We realized later that this copy is needed only when the user's buffer is to be filled noncontiguously. In the contiguous case, data can be received directly into the appropriate location in the user's buffer. We similarly experimented with different ways of communicating data in MPI and measured the effect on overall collective-I/O performance with different MPI implementations and on different machines. We selected nonblocking communication with the receives posted first and then the sends, which performs the best on most systems. It may be possible, however, to tune the communication further on some machines by posting the sends before the receives or by using MPI's persistent requests.

#### 4.2.4 Portability Issues

We were able to implement these optimizations portably and without sacrificing performance by using ADIO as a portability layer for I/O (see Section 2) and by using MPI for communication. Data sieving and collective I/O are implemented within ADIO functions [13]; data sieving is used in ADIO functions that read/write noncontiguous data, and collective I/O is used in ADIO's collective-I/O functions. Both these optimizations ultimately make contiguous I/O requests to the underlying file system, which are implemented by using ADIO's contiguous-I/O functions. The contiguous-I/O functions, in turn, are implemented using the appropriate file-system call for each different file system.

## 5 Performance Measurements

We used three applications for our performance experiments:

1. DIST3D, a template representing the I/O access pattern in an astrophysics application (ASTRO3D) from the University of Chicago;
2. the NAS BTIO benchmark [4]; and
3. an unstructured code (UNSTRUC) written by Larry Schoof and Wilbur Johnson of Sandia National Laboratories.

The I/O in DIST3D consists of reading/writing a three-dimensional array distributed in a (block,block,block) fashion among processes from/to a file containing the global array in row-major order. The BTIO benchmark [4] simulates the I/O required by a time-stepping flow solver that periodically writes its solution matrix. The benchmark only performs writes, but we modified it to perform reads also. UNSTRUC emulates the I/O access pattern in unstructured-grid applications by generating a random irregular mapping from the local one-dimensional array of a process to a global array in a common file shared by all processes. The mapping specifies where each element of the local array is located in the global array.

We ran the code portably and measured the performance on five different parallel machines: the HP Exemplar and SGI Origin2000 at the National Center for Supercomputing Applications (NCSA), the IBM SP at Argonne National Laboratory, the Intel Paragon at California Institute of Technology, and the NEC SX-4 at the National Aerospace Laboratory (NLR) in Holland. We used the native parallel 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. At the time we performed the experiments, these file systems were configured as follows: HFS on the Exemplar was configured on twelve disks; XFS on the Origin2000 had two RAID units with SCSI-2 interfaces; the SP had four servers for PIOFS, and each server had four SSA disks attached to it in one SSA loop; the Paragon had 64 I/O nodes for PFS, each with an individual Seagate disk; and SFS on the NEC SX-4 was configured on a single RAID unit comprising sixteen SCSI-2 data disks.

We measured the I/O performance of these applications by using MPI-IO functions to perform I/O in three different ways as follows:

**Unix-style accesses** Separate MPI-IO function calls to access each individual contiguous piece of data.

**Data sieving** Create a file view to describe a noncontiguous access pattern and use a single *independent* MPI-IO function to access data.

**Collective I/O** Create a file view to describe a noncontiguous access pattern and use a single *collective* MPI-IO function to access data.

In all experiments, we used the default buffer sizes for data sieving and collective I/O (see Sections 3 and 4) and the default values of the file-striping parameters on all file systems.

Tables 1 and 2 show the read and write bandwidths for DIST3D. The performance of Unix-style accesses was, in general, very poor. By using data sieving instead, the read bandwidth improved by a factor ranging from 2.6 on the HP Exemplar to 453 on the NEC SX-4. The write bandwidth improved by a factor ranging from 2.3 on the HP Exemplar to 121 on the NEC SX-4. Data sieving cannot be performed for writing on the SP's PIOFS file system, because PIOFS does not support file locking. On PIOFS, ROMIO therefore translates noncontiguous, independent write requests into multiple Unix-style accesses.

The performance improvement with collective I/O was much more significant. The read bandwidth improved by a factor of as much as 793 over Unix-style accesses (NEC SX-4) and as much as 14 over data sieving (Intel Paragon). The write performance improved by a factor of as much as

**Table 1. Read performance of DIST3D (array size 512x512x512 integers = 512 Mbytes)**

| | | Bandwidth (Mbytes/s) | | |
|---|---|---|---|---|
| Machine | Proc-essors | Unix-style | Data Sieving | Coll-ective |
| HP Exemplar | 64 | 5.42 | 14.2 | 68.2 |
| IBM SP | 64 | 2.13 | 11.9 | 90.2 |
| Intel Paragon | 256 | 3.01 | 9.50 | 132 |
| NEC SX-4 | 8 | 0.71 | 322 | 563 |
| SGI Origin2000 | 32 | 14.0 | 118 | 175 |

**Table 2. Write performance of DIST3D (array size 512x512x512 integers = 512 Mbytes)**

| | | Bandwidth (Mbytes/s) | | |
|---|---|---|---|---|
| Machine | Proc-essors | Unix-style | Data Sieving | Coll-ective |
| HP Exemplar | 64 | 0.54 | 1.25 | 50.7 |
| IBM SP | 64 | 1.85 | N/A | 57.6 |
| Intel Paragon | 256 | 1.12 | 3.33 | 183 |
| NEC SX-4 | 8 | 0.62 | 75.3 | 447 |
| SGI Origin2000 | 32 | 5.06 | 13.1 | 66.7 |

721 over Unix-style accesses (NEC SX-4) and as much as 40 over data sieving (HP Exemplar).

Tables 3 and 4 present results for Class C of the BTIO benchmark. (BTIO requires the number of processors to be a perfect square.) For BTIO, Unix-style accesses performed better than data sieving on three out of the five machines. The reason is that the holes between data segments needed by a process are large in BTIO—more than five times the size of the data segment. As a result, a lot of unwanted data was accessed during data sieving, resulting in lower performance than with Unix-style accesses. As mentioned in Section 3, an intelligent data-sieving algorithm can detect such large holes and internally perform Unix-style accesses. ROMIO's data sieving algorithm does not currently do this, however.

Collective I/O performed extremely well on BTIO, because no unwanted data was accessed during collective I/O and all accesses were large. The performance improved by a factor of as much as 512 over Unix-style accesses for reading and 597 for writing, both on the NEC SX-4.

Tables 5 and 6 show the read and write bandwidths for UNSTRUC. In this application, the I/O access pattern is irregular, and the granularity of each access is very small (64 bytes). Unix-style accesses are not feasible for this kind

**Table 3. Read performance of BTIO (Class C, problem size 5x162x162x162 double precision $\approx$ 162 Mbytes)**

| | | Bandwidth (Mbytes/s) | | |
|---|---|---|---|---|
| Machine | Proc-essors | Unix-style | Data Sieving | Coll-ective |
| HP Exemplar | 64 | 6.35 | 5.84 | 44.2 |
| IBM SP | 64 | 2.73 | 1.66 | 80.6 |
| Intel Paragon | 256 | 2.28 | 1.23 | 82.0 |
| NEC SX-4 | 9 | 1.26 | 116 | 645 |
| SGI Origin2000 | 36 | 12.1 | 37.0 | 107 |

**Table 4. Write performance of BTIO (Class C, problem size 5x162x162x162 double precision $\approx$ 162 Mbytes)**

| | | Bandwidth (Mbytes/s) | | |
|---|---|---|---|---|
| Machine | Proc-essors | Unix-style | Data Sieving | Coll-ective |
| HP Exemplar | 64 | 0.86 | 0.50 | 29.7 |
| IBM SP | 64 | 2.21 | N/A | 38.6 |
| Intel Paragon | 256 | 1.37 | 0.45 | 98.8 |
| NEC SX-4 | 9 | 0.99 | 29.9 | 591 |
| SGI Origin2000 | 36 | 7.93 | 2.90 | 67.2 |

of application, as they take an excessive amount of time. We therefore do not present results for Unix-style accesses for UNSTRUC. Collective I/O again performed much better than independent I/O with data sieving, the only exception being for reads on the NEC SX-4. In this case, because of the high read bandwidth of NEC's Supercomputing File System (SFS), data sieving by itself outperformed the extra communication required for collective I/O.

## 6 Conclusions

For parallel applications to achieve high I/O performance, it is critical that the parallel-I/O system be able to deliver high performance even for noncontiguous access patterns. We have described two optimizations our MPI-IO implementation performs that enable it to deliver high performance even if the user's request consists of many small, noncontiguous accesses. Our implementation of these optimizations generalizes the work in [11, 12] to handle *any* noncontiguous access pattern, not just sections of arrays.

For the applications we considered, collective I/O performed significantly better than both data sieving and

**Table 5. Read performance of UNSTRUC**

| Machine | Proc-essors | Grid Points | Bandwidth (Mbytes/s) Data Sieving | Coll-ective |
|---|---|---|---|---|
| HP Exemplar | 64 | 8 million | 3.15 | 35.0 |
| IBM SP | 64 | 8 million | 1.63 | 73.3 |
| Intel Paragon | 256 | 8 million | 1.18 | 78.4 |
| NEC SX-4 | 8 | 8 million | 152 | 101 |
| SGI Origin2000 | 32 | 4 million | 30.0 | 80.8 |

**Table 6. Write performance of UNSTRUC**

| Machine | Proc-essors | Grid Points | Bandwidth (Mbytes/s) Data Sieving | Coll-ective |
|---|---|---|---|---|
| HP Exemplar | 64 | 8 million | 0.18 | 22.1 |
| IBM SP | 64 | 8 million | N/A | 37.8 |
| Intel Paragon | 256 | 8 million | 0.22 | 94.9 |
| NEC SX-4 | 8 | 8 million | 16.8 | 81.5 |
| SGI Origin2000 | 32 | 4 million | 1.33 | 59.2 |

Unix-style accesses. Data sieving performed much better than Unix-style accesses for DIST3D and UNSTRUC. For BTIO, on some machines, Unix-style accesses performed better than data sieving, because of large holes between data segments accessed by each process in BTIO.

The implementation of data sieving and collective I/O must be carefully tuned to minimize the overhead of buffer copying and interprocess communication. Otherwise, these overheads can impact performance significantly.

## Acknowledgments

We thank Larry Schoof and Wilbur Johnson for providing the unstructured code used in this paper. This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38; and by the Scalable I/O Initiative, a multiagency project funded by the Defense Advanced Research Projects Agency (contract number DABT63-94-C-0049), the Department of Energy, the National Aeronautics and Space Administration, and the National Science Foundation.

## References

[1] S. Baylor and C. 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.

[2] P. Crandall, R. Aydt, A. Chien, and D. Reed. Input-Output Characteristics of Scalable Parallel Applications. In *Proceedings of Supercomputing '95*. ACM Press, December 1995.

[3] J. del Rosario, R. Bordawekar, and A. 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.

[4] S. Fineberg, P. Wong, B. Nitzberg, and C. 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.

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

[6] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*. July 1997. On the World-Wide Web at `http://www.mpi-forum.org/docs/docs.html`.

[7] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. Ellis, and M. Best. File-Access Characteristics of Parallel Scientific Workloads. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1075–1089, October 1996.

[8] 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.

[9] E. Smirni, R. Aydt, A. Chien, and D. 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.

[10] E. Smirni and D. Reed. Lessons from Characterizing the Input/Output Behavior of Parallel Scientific Applications. *Performance Evaluation: An International Journal*, 33(1):27–44, June 1998.

[11] R. Thakur and A. Choudhary. An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays. *Scientific Programming*, 5(4):301–317, Winter 1996.

[12] R. Thakur, A. Choudhary, R. Bordawekar, S. More, and S. Kuditipudi. Passion: Optimized I/O for Parallel Applications. *Computer*, 29(6):70–78, June 1996.

[13] R. Thakur, W. Gropp, and E. Lusk. An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. In *Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation*, pages 180–187. IEEE Computer Society Press, October 1996.

[14] R. Thakur, W. Gropp, and E. 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.