A Case for Using MPI's Derived Datatypes to Improve I/O Performance*

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

Preprint ANL/MCS-P717-0598, May 1998

Submitted as an extended abstract to
SC98: High Performance Networking and Computing

Abstract

MPI-IO, the I/O part of the MPI-2 standard, is a promising new interface for parallel I/O. A key feature of MPI-IO is that it allows users to access several noncontiguous pieces of data from a file with a single I/O function call by defining file views with derived datatypes. We explain how critical this feature is for high performance, why users must create and use derived datatypes whenever possible, and how it enables implementations to perform optimizations. In particular, we describe two optimizations our MPI-IO implementation, ROMIO, performs: data sieving and collective I/O. We present performance results on five different parallel machines: HP Exemplar, IBM SP, Intel Paragon, NEC SX-4, and SGI Origin2000.

*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.

1 Introduction

I/O is a major bottleneck in many parallel applications. One of the main reasons for poor I/O performance is that most parallel file systems provide a Unix-like application program interface (API) for parallel I/O. The Unix API allows a user to access only a single, contiguous chunk of data at a time from a file.[1] Such an API cannot concisely express the I/O access patterns common in parallel programs, namely, a number of relatively small, noncontiguous accesses, and it also cannot express the notion of multiple processes making *collective* requests. Consequently, the I/O system is constrained in the optimizations it can perform. Furthermore, many parallel file systems provide their own extensions to or variations of the traditional Unix API, which make programs nonportable.

To overcome the performance and portability limitations of existing parallel I/O APIs, the MPI Forum defined an interface for parallel-I/O (which we call MPI-IO) as part of the MPI-2 standard [6]. MPI-IO is a rich API with many features designed specifically for performance and portability, such as support for noncontiguous accesses, collective I/O, nonblocking I/O, and a standard data representation. We believe that MPI-IO has the potential to solve many of the problems users currently face with parallel I/O.

One way to port a Unix-I/O program to MPI-IO is to replace all Unix-I/O functions with their MPI-IO equivalents. Such a port is easy but will very likely not improve performance. To get real performance benefits with MPI-IO, users must use some of MPI-IO's advanced features. In this paper, we focus on a key feature of MPI-IO: the ability to access noncontiguous data with a single I/O function call by defining file views with MPI's derived datatypes. We describe how derived datatypes allow users to express compactly and portably the entire I/O access pattern in their application. We explain how critical it is that users create and use these derived datatypes and how such use enables an MPI-IO implementation to perform optimizations. In particular, we describe two optimizations our implementation, ROMIO [13], performs: data sieving and collective I/O. We use a distributed-array example and an unstructured code to illustrate the performance improvements these optimizations provide.

2 Derived Datatypes and MPI-IO

In MPI-1, all message-passing functions have a *datatype* argument [5]. 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 different kinds of datatype constructors in MPI are as follows:

- **contiguous** Creates a new datatype consisting of contiguous copies of an old one.
- **vector/hvector** Creates a new datatype consisting of equally spaced copies of an old one.
- **indexed/hindexed/indexed_block** Allows replication of a datatype into a sequence of blocks, each containing multiple copies of the old datatype; the blocks may be unequally spaced.
- **struct** The most general datatype constructor, which allows each block to consist of replications of different datatypes.
- **subarray** Creates a datatype that corresponds to a subarray of a multidimensional array.
- **darray** Creates a datatype that describes a process's local array obtained from a regular distribution a multidimensional global array.

Of these datatype constructors, indexed_block, subarray, and darray were added in MPI-2; the others were defined in MPI-1.

The datatype created by a datatype constructor can be used as an input datatype to another datatype constructor. Therefore, any noncontiguous data layout can be represented in terms of a derived datatype.

[1] 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 noncontiguity in the file, but it is not supported on all file systems and is not collective.

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 a file. The former is the same as in MPI message-passing functions and can be used, for example, when the user's buffer represents a local array with a "ghost area" that is not to be written to the file. The latter, a new feature in MPI-IO, is the mechanism a process uses to describe the portion of a file it wants to access, also called a *file view*. The file view can be defined by using any MPI basic or derived datatype; therefore, any general, noncontiguous access pattern can be compactly represented.

Several studies have shown that, in many parallel applications, each process needs to access a number of relatively small, noncontiguous portions of a file [7, 2, 12, 1, 9]. From a performance perspective, it is critical that the I/O API can express such an access pattern, as it enables the implementation to optimize the I/O request. The optimizations typically allow the physical I/O to take place in large, contiguous chunks, even though the user's request may be noncontiguous. MPI-IO's file views, therefore, are critical for performance. Users must ensure that they describe noncontiguous access patterns in terms of a file view and then call a single I/O function; they must not try to access each contiguous portion separately as in Unix I/O.

3 A Classification of I/O Request Structures

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 what I/O functions the application uses and how. We classify the different ways of expressing I/O access patterns in MPI-IO into four "levels," named level 0–level 3. We explain this classification with the help of an example, accessing a distributed array from a file, which is a common access pattern in parallel applications.

Consider a two-dimensional array distributed among 16 processes in a (block, block) fashion as shown in Figure 1. 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 located noncontiguously in the file.

Figure 2 shows four ways in which a user can express this access pattern in MPI-IO, which we name level 0–level 3. 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 also represent increasing amounts of data per request, as illustrated in Figure 3.[2] It is well known that larger the size of an I/O request, higher is the performance. Therefore, users must strive to express their I/O requests as level 3 rather than level 0. How good the performance is at each level depends on how well the implementation takes advantage of the extra access information at each level.

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. Many real, parallel applications, however, do not fall under this category [7, 2, 12, 1, 9].

4 Optimizations

We describe some of the optimizations an MPI-IO implementation can perform when the user uses derived datatypes to specify noncontiguous accesses. The first two optimizations, data sieving and collective I/O, are already implemented in ROMIO [13].

[2] 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.

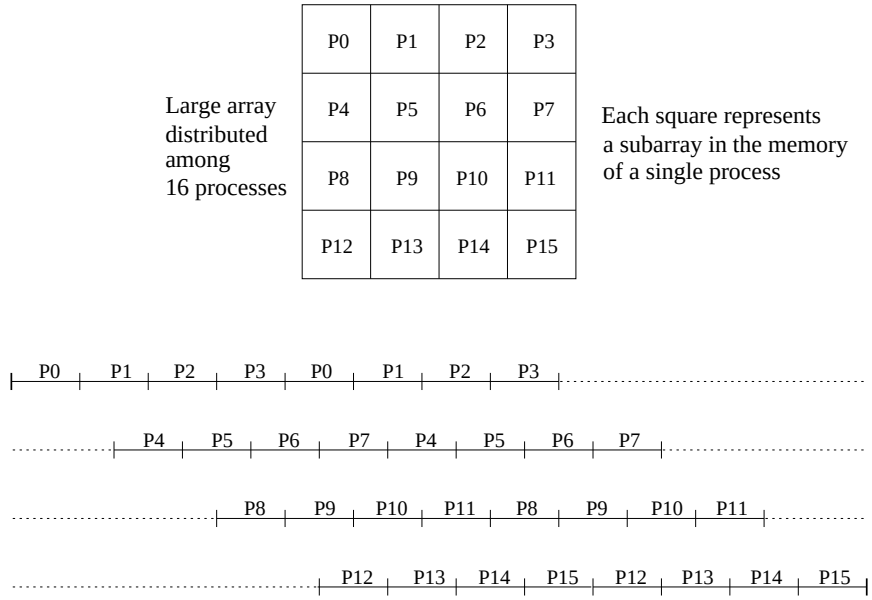

Figure 1: Distributed-array access

```
MPI_File_open(file, ..., &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(file, ..., &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(file, ..., &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(file, ..., &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 2: Pseudo-code that shows four ways of accessing the data in Figure 1 with MPI-IO

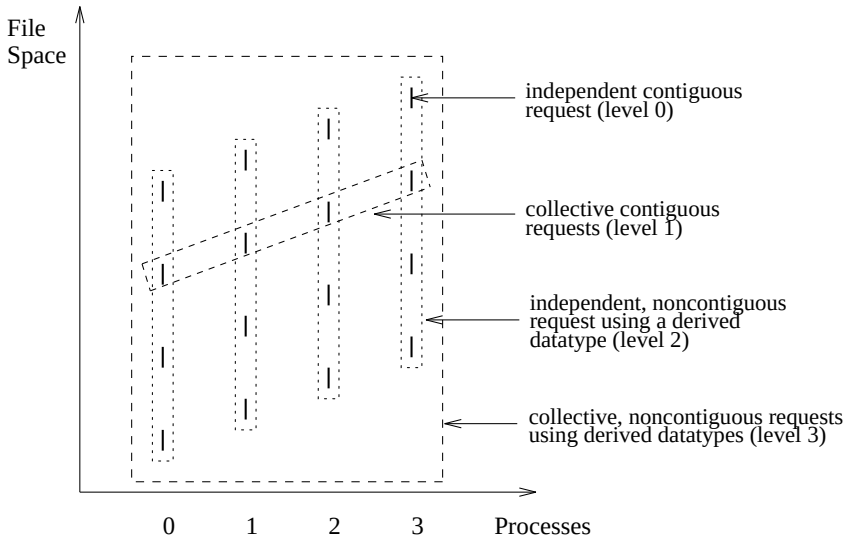

Figure 3: The four levels representing increasing amounts of data per request

4.1 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. Data sieving [11] 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. The basic idea in data sieving is to make large I/O requests and extract, in memory, the data that is really needed. When the user makes a read request for noncontiguous data, ROMIO reads large, contiguous chunks, starting from the first requested byte in the file, into a temporary buffer in memory and then copies the requested portions into the user's buffer. More data is read than is actually needed, but the benefit of reading large, contiguous chunks far outweighs the cost of reading unwanted data (see Section 5). The intermediate buffering requires extra memory, but ROMIO uses only a constant amount of extra memory that does not increase with the size of the user's request. Furthermore, the user can control at run time the amount of extra memory ROMIO uses, via MPI-IO's hints mechanism.

Noncontiguous write requests are handled similarly, except that ROMIO must perform a read-modify-write to avoid destroying the data in the gaps between the portions the user actually wants to write. In the case of independent write requests, during the read-modify-write, ROMIO must also lock the corresponding portion of the file, because other processes may independently try to access portions that are interleaved with this access.

4.2 Collective I/O

The collective-I/O functions in MPI-IO must be called by all processes that together opened the file. This property enables the implementation to analyze and merge the requests of different processes. In many cases, the merged request may be large and contiguous, although the individual requests were noncontiguous. The merged request can, therefore, be serviced efficiently [3, 10, 4, 8].

ROMIO has an optimized implementation of collective I/O that uses a generalized version of the extended two-phase method described in [10]. The basic idea is to perform I/O in two phases: an I/O phase and a communication phase. In the I/O phase, processes perform I/O for the merged request. If the merged request is not contiguous by itself, data sieving is used to obtain contiguous accesses. In the communication phase, processes redistribute data among themselves to achieve the desired distribution. For reading, the first phase is the I/O phase, and the second phase is the communication phase. For writing, it is the reverse. The additional cost of the communication phase is negligible compared with the benefit obtained by performing I/O contiguously. As in data sieving, ROMIO uses a constant amount of additional memory for performing

collective I/O, which can be controlled by the user at run time. The user can also control the number of processes that perform I/O in the I/O phase, which is useful on systems where the I/O performance does not scale with the number of processes making concurrent requests. To enable maximum amount of merging and larger accesses, the user's collective request must be a level-3 request rather than a level-1 request.

4.3 Improved Prefetching and Caching

When the user specifies complete access information in a single I/O function call, the MPI-IO implementation or file system does not need to guess what the future accesses will be. It can, therefore, perform better prefetching and caching. (This optimization is not yet implemented in ROMIO.)

5 Performance Results

We used two applications to measure the effect of using derived datatypes on performance: the distributed array example of Figure 1 and an unstructured code we obtained from Larry Schoof and Wilbur Johnson of Sandia National Laboratories. (Details of the unstructured code will be provided in the final version of the paper.) We modified the I/O portions of these applications to correspond to each of the four levels of requests and ran the programs on five different parallel machines—HP Exemplar, IBM SP, Intel Paragon, NEC SX-4, and SGI Origin2000—using ROMIO as the MPI-IO implementation.

If the requests of processes that call a collective-I/O function are not interleaved in the file, ROMIO's collective-I/O implementation just calls the corresponding independent-I/O function on each process. For the distributed-array example, therefore, level-1 requests perform the same as level-0 requests. However, if the accesses in a level-1 request are interleaved or overlapping (e.g., in a read-broadcast type of access pattern), ROMIO implements the level-1 request collectively, and the performance is better than with a level-0 request.

For level-2 requests, ROMIO performs data sieving. Depending on the machine, we observed an improvement ranging from 13.2% (IBM SP) to 45,252% (NEC SX-4) for the distributed-array example (see Table 1). In the unstructured code, the I/O access pattern is irregular, and the granularity of each access is very small. Level-0 requests are not feasible for this kind of application, as they take an inordinate amount of time. Therefore, we do not present level-0/1 results for the unstructured code. Level-2 requests performed reasonably well for the unstructured code, but not as well as level-3 requests (see Table 2).

ROMIO performs collective I/O in the case of level-3 requests. Within the collective-I/O implementation, ROMIO also performs data sieving when there are holes in the merged request. In these two examples, however, the merged request had no holes. Collective I/O improved the performance of level-3 requests significantly. For the distributed-array example, the improvement was as high as 79,196% over level-0/1 requests (NEC SX-4) and as high as 1,289% over level-2 requests (Intel Paragon). Similarly, for the unstructured code, the improvement was as high as 5,681% over level-2 requests (Intel Paragon). An unusual result was observed on the NEC SX-4 for the unstructured code: Level 2 performed better than level 3. We attribute it to the high read bandwidth of NEC's Supercomputing File System (SFS), due to which data sieving by itself outperformed the extra communication required to merge requests in collective I/O.

Detailed performance and scalability results, including write bandwidths, will be provided in the final version of the paper.

6 Conclusions

MPI-IO has the potential to help users achieve better I/O performance in parallel applications. On their part, users must use some of the special features of MPI-IO. In particular, when accesses are noncontiguous, users must strive to create derived datatypes and define file views. Our results show that performance improves by orders of magnitude when the users create derived datatypes and use the collective-I/O functions.

We note that the MPI-IO standard does not *require* an implementation to perform any of these optimizations. However, even if an implementation does not perform any optimization and instead translates level-3

Table 1: I/O performance for distributed-array access (array size $512 \times 512 \times 512$ integers, file size 512 MB)

Machine	Number of Processors	Read Bandwidths (MB/s) Level 0/1	Level 2	Level 3
HP Exemplar	64	5.42	14.2	68.2
IBM SP	48	5.83	6.60	88.4
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: I/O performance of an unstructured code

Machine	Number of Processors	Number of Grid Points	Read Bandwidths (MB/s) Level 2	Level 3
HP Exemplar	64	8 million	4.37	37.1
IBM SP	64	8 million	1.64	74.1
Intel Paragon	256	8 million	1.28	74.0
NEC SX-4	8	8 million	97.3	56.0
SGI Origin2000	32	4 million	43.8	123

requests into several level-0 requests to the file system, the performance would be no worse than if the user made level-0 requests himself.

In the final version of the paper, we will provide more details on the optimizations and more performance and scalability results.

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

[5] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. Version 1.1, June 1995. On the World-Wide Web at `http://www.mpi-forum.org/docs/docs.html`.

[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] 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.

[11] 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.

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

[13] R. Thakur, E. Lusk, and W. Gropp. Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation. Technical Report ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, October 1997.