Noncontiguous I/O Accesses Through MPI-IO

Avery Ching Alok Choudhary Kenin Coloma Wei-keng Liao
Center for Parallel and Distributed Computing
Northwestern University
Evanston, IL 60208
{aching, choudhar, kcoloma, wkliao}@ece.nwu.edu

Robert Ross William Gropp
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
{rross, gropp}@mcs.anl.gov

Abstract

I/O performance remains a weakness of parallel computing systems today. While this weakness is partly attributed to rapid advances in other system components, I/O interfaces available to programmers and the I/O methods supported by file systems have traditionally not matched efficiently with the types of I/O operations that scientific applications perform, particularly noncontiguous accesses. The MPI-IO interface allows for rich descriptions of the I/O patterns desired for scientific applications and implementations such as ROMIO have taken advantage of this ability while remaining limited by underlying file system methods.

A method of noncontiguous data access, list I/O, was recently implemented in the Parallel Virtual File System (PVFS). We implement support for this interface in the ROMIO MPI-IO implementation. Through a suite of noncontiguous I/O tests we compared ROMIO list I/O to current methods of ROMIO noncontiguous access and found that the list I/O interface provides performance benefits in many noncontiguous cases.

1. Introduction

It has been known for some time that scientific applications tend to access files in a noncontiguous manner [13, 2, 7]. Accordingly, a number of I/O optimizations, in particular data sieving and two phase I/O, were developed to address noncontiguous access patterns in situations where only contiguous I/O operations were available [8]. With the introduction of MPI-IO, a standard interface for I/O better suited to scientific applications was made available [17]. These same optimizations were implemented in the ROMIO MPI-IO implementation, which has become the most popular MPI-IO implementation to date [18].

In [19], Thakur et al. noted that the POSIX I/O interface [10] available for accessing most file systems was not an ideal interface for addressing noncontigous I/O on parallel file systems and proposed an interface for describing noncontiguous reiogns in memory and file that would serve as a flexible API for noncontiguous I/O to a parallel file system. The goal of this interface was to provide the MPI-IO implementation with an efficient means of noncontiguous I/O. In [6], Ching et al. implemented support for this API, denoted *list I/O*, in the Parallel Virtual File System (PVFS) [4].

This work describes how we leverage this capability through the ROMIO MPI-IO implementation and compares it to the other access methods available through ROMIO. By enhancing ROMIO to use list I/O, we provide the performance benefits of this new feature to applications without changing how the application itself performs I/O. In Section 2 we describe the list I/O interface and how ROMIO can use this interface to its advantage. In Section 3 we compare the ROMIO list I/O approach with previous ROMIO I/O methods. In Section 4 we observe the performance implications of list I/O support for three applications: a tile reader application, a ROMIO three-dimensional block test, and a simulation of FLASH I/O checkpointing. In Section 5 we discuss related work in the area of noncontiguous I/O. In Section 6 we conclude our work and point to areas of future study.

2. List I/O

The list I/O interface proposed in [19] by Thakur et al. is an I/O interface for describing noncontiguous data in memory and in file. It is a simple interface that can describe complex noncontiguous data access in a single function call. We present a visual example of the list I/O interface in Figure 1. Below is the list I/O interface proposed:

```
list_io_read(int   mem_list_count,
             char *mem_offsets[],
             char  mem_lengths[],
             int   file_list_count,
             int   file_offsets[],
             int   file_lengths[])
```

(list_io_write has the same parameters)

- *mem_list_count* is the total number of contiguous memory regions involved in the data access, which is also the length of the arrays *mem_offsets[]* and *mem_lengths[]*.

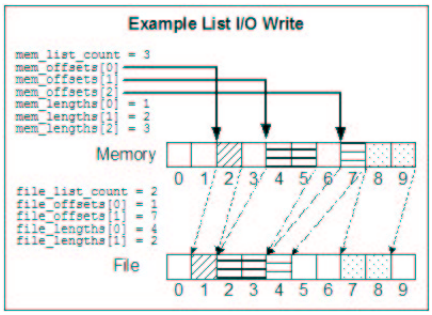

Figure 1: Example list I/O write. Since only contiguous regions can be described using the POSIX read/write interfaces, four I/O calls would be required instead of one list I/O write.

- *mem_offsets[]* is an array of pointers that each point to the beginning of a contiguous memory region.
- *mem_lengths[]* is an array of lengths that match every start of a contiguous memory region with a corresponding memory length.
- *file_list_count* has the same functionality for a file as *mem_list_count* does for memory. It is the total number of contiguous file regions as well as the length of the arrays *file_offsets[]* and *file_lengths[]*.
- *file_offsets[]* is an array of offsets that each point to the beginning of a contiguous file region.
- *file_lengths[]* is the lengths of the file regions that correspond to the file offsets. The sum of the *mem_lengths[]* and *file_lengths[]* must be equivalent.

A naive implemention of list I/O would use the POSIX read/write calls and would provide no performance advantage over those calls. However, building support directly into the parallel file system to handle such a call provides the file system with much needed noncontigous I/O capabilities. An example of a high performance implementation of list I/O was added to the Parallel Virtual File System (PVFS) and will be discussed further in Section 2.1.

Traditional methods of noncontiguous data access include multiple contiguous I/O calls or the use of data sieving. Multiple contiguous I/O builds on traditional POSIX I/O calls (read/write) to perform noncontiguous access. Data sieving is the I/O optimization of reading a large contiguous amount of data from file into a memory buffer, again building on the POSIX API, and performing all noncontiguous data movement using the memory buffer.

Using list I/O for noncontiguous data access offers several advantages over traditional methods. Multiple contiguous I/O calls have a large overhead with respect to the number of I/O calls that must be issued to the underlying file system when describing complex noncontiguous I/O access patterns. List I/O can perform a noncontiguous I/O data access with fewer I/O calls with an optimized implementation. Data sieving requires a read-modify-write set of operations and file synchronization (which often has prohibitive overhead) in noncontiguous writes. Some file systems do not provide synchronization and therefore cannot support data sieving in the noncontiguous write case. Also, unlike other noncontiguous methods, data sieving requires memory for the data sieving buffer. List I/O provides a simple interface that can provide high performance I/O for reading and writing data in scientific workloads.

Section 2.1 describes PVFS and its implementation of list I/O. Section 2.2 discusses ROMIO, the MPI-IO implementation developed at Argonne National Laboratory, and how ROMIO uses the list I/O interface in its Abstract-Device implementation for I/O (ADIO) implementation.

2.1 Parallel Virtual File System

The Parallel Virtual File System (PVFS) is a client-server parallel file system that provides parallel data access with a clusterwide consistent name space. PVFS addresses the need for high performance I/O on low-cost Linux clusters. ROMIO, the MPI-IO implementation from Argonne National Laboratory, has native support for PVFS. Additional information on PVFS can be found in [4], and a performance evaluation on a Linux cluster can be found in [1].

Ching et al. created support for list I/O in PVFS [6]. The `pvfs_read_list` and `pvfs_write_list` functions take list I/O parameters and perform the noncontiguous access as a single PVFS operation. The maximum number of file offset-length pairs serviced by a single I/O request is capped at 128 file regions to limit request sizes across the network, and the implementation will transparently divide larger collections into multiple requests. The reduction of the number of I/O requests over the traditional multiple contiguous I/O method leads to higher performance data access in several I/O intensive benchmarks. The addition of list I/O to the PVFS client library adds a high performance method of noncontiguous data access; however, few users directly use the PVFS client library for I/O. It is important that support through a common API be established if this feature is to be used; ROMIO is the obvious candidate for providing this support.

2.2 ROMIO MPI-IO Implementation

ROMIO is an implementation of the MPI-2 I/O specification built on top of the MPI-1 message-passing operations and ADIO, a small set of basic functions for performing I/O [18, 16]. ROMIO can support any file system by implementing the ADIO functions with the interfaces of the file system. Since each file system can implement the ADIO functions using its own file-system library, ROMIO can take full advantage of most file-system specific optimizations. ROMIO already has support for many file systems including HP's HFS, NFS, Intel's PFS, SGI's XFS, Unix FS, and PVFS. It is used for I/O in many MPI implementations, including MPICH and LAM/MPI.

```
MPI_File_open(MPI_COMM_WORLD,
              ''/pvfs/test.txt'',
              MPI_MODE_RDWR,
              MPI_INFO_NULL, &fh);
etype = MPI_INT;
MPI_Type_vector(2, 2, 3,
                MPI_INT,
                &filetype);
MPI_Type_commit(&filetype);
MPI_File_set_view(fh, 0, etype,
                  filetype, datarep,
```

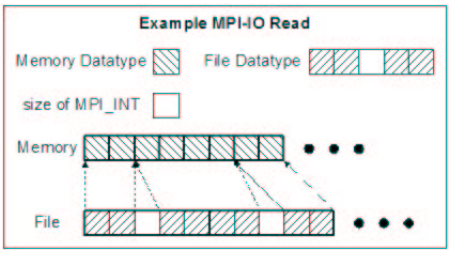

Figure 2: File datatypes are replicated and read into memory until the read call has accessed that correct amount of data.

```
                    MPI_INFO_NULL);
MPI_File_read(fh, buf, 8,
              MPI_INT, status,
              ierror);
MPI_File_close(&fh);
```

This example MPI-IO C code, as graphically depicted in Figure 2, performs a collective open of a file with `MPI_File_open`. MPI Type calls are used to create both the memory datatype and the file datatype. Performing the read puts the first two integers from file into the memory buffer, then puts the fourth and fifth integers from file into the memory buffer, continuing the vector pattern of the file datatype until eight `MPI_INTS` are contiguously located in the buffer `buf`.

To take advantage of PVFS's list I/O in the ROMIO MPI-IO implementation, we implemented the new ADIO read and write functions using `pvfs_read_list` and `pvfs_write_list`. PVFS list I/O, as described in Section 2, requires several parameters, including offset-length pairs for memory locations, offset-length pairs for file locations, and their respective counts. Our new ADIO calls convert MPI types into their respective contiguous regions and create the arrays to pass to the list I/O calls. This conversion process, or *flattening*, is accomplished by decoding the types using the MPI calls `MPI_Type_get_envelope` and `MPI_Type_get_contents`. Figure 3 gives an example of this process. The implementation for generating list I/O calls in ROMIO breaks up the file regions into collections of 128 to match the maximum size allowable by the PVFS list I/O implementation. Given a large number of noncontiguous accesses, ROMIO will fill the offset-length arrays, perform the required I/O and repeat the sequence until all the noncontiguous regions have been satisfied.

3 Comparison of ROMIO Implementations

In this section we describe the other I/O methods implemented in ROMIO. Section 3.1 discusses the ROMIO implementation over POSIX read/write calls, which we call *ROMIO POSIX I/O*. Section 3.2 covers the data sieving implementation in ROMIO, which we call *ROMIO data sieving I/O*. Section 3.3 explains the two phase collective implementation, which we call *ROMIO collective I/O*.

3.1 ROMIO POSIX I/O

In ROMIO POSIX I/O, all noncontiguous data access is reduced to multiple POSIX [10] read/write calls on contiguous file regions. This implementation is generally

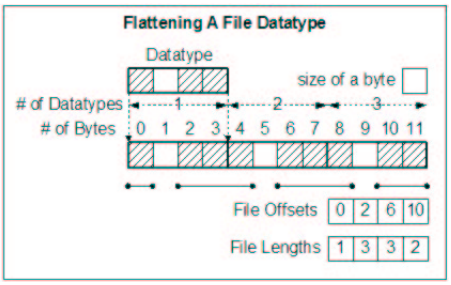

Figure 3: Example flattening of a file datatype. File datatypes are converted into lists of file offsets and lengths in order to generate the necessary parameters to use the list I/O interface.

turned off in favor of ROMIO data sieving I/O in all read cases. For noncontiguous writes when the underlying file system does not support file locking, ROMIO data sieving I/O will not work properly and ROMIO POSIX I/O must be used in order to get semantically consistent results.

3.2 ROMIO Data Sieving I/O

ROMIO data sieving I/O uses a data sieving [3] method to reduce the number of I/O calls to the underlying file system. Data sieving works by reading large contiguous chunks of file data at a time into a data sieving buffer in memory and then extracting the desired regions. In the case of noncontiguous writes, large contiguous chunks of file data are read into memory, new data is written to the data sieving buffer, and finally the data sieving buffer is contiguously written back to the file. This process can reduce the number of I/O calls to the file system by performing data movement in the data sieving buffer instead of at the disk. We tested ROMIO data sieving I/O with a data sieving buffer of 4 MBytes, the default size.

In order to use ROMIO data sieving I/O in noncontiguous writes, the file system must support file locking to maintain correctness. However, since PVFS does not support file locking, ROMIO invokes the ROMIO POSIX I/O method instead.

3.3 ROMIO Collective I/O

ROMIO collective I/O is an optimization for aggregation of reads/writes using the two phase I/O method developed in [8]. Other examples of collective I/O are disk directed I/O [12] and server directed I/O [15]. Two phase I/O works by assigning file regions to a specified number of processors who will handle I/O on behalf of all processes. In the ROMIO implementation these regions are dynamically calculated based on the extent of the regions being accessed by all processes, with the entire region being split into evenly sized contiguous sub-regions. Before a collective read operation, the processors designate file partitions and pass file offset-length pairs to assigned I/O processors. The first phase of the two-phase method is parallel data sieving reads of the aggregate file region by the I/O processors and the second phase is an exchange of data between the I/O processors and the processors who had file regions in the I/O processor's file partition. In a collective write operation, the first phase of the two-phase method is an exchange of data from the relevant processors to the assigned

Table 1: I/O Characteristics of the Tile Reader Benchmark

	Desired data per client	Data accessed per client	# of I/O ops per client	Resent data per client
POSIX I/O	2.25 MB	2.25 MB	768	—
Data Sieving I/O	2.25 MB	5.56 MB	2	—
Collective I/O	2.25 MB	5.56 MB	1	4.50 MB
List I/O	2.25 MB	2.25 MB	6	—

I/O processors and the second phase is parallel writes to the file partitions by the I/O processors. The write case of ROMIO collective I/O does not use data sieving writes from the I/O processors to the file system since there is no file synchronization in PVFS. For a file system that supports file locking, the second phase of the write case of ROMIO collective I/O would use data sieving writes.

4 Performance Implications

To show the effect of the ROMIO list I/O optimization, we ran a series of tests that access noncontiguous data. Section 4.1 discusses our test setup. We tested our optimization using a tiled reader benchmark, a three-dimensional block benchmark from the ROMIO testing suite, and a simulation of the I/O portion of the FLASH code. In each section we provide a table summarizing the I/O characteristics of the application for each access method. This information helps in explaining the performance of the methods. The values for data accessed per client and resent data per client are average values across processes.

4.1 Machine Configuration

We ran all of our tests on the Chiba City cluster at Argonne National Laboratory [5]. The cluster had the following configuration at test time. There are 256 nodes each with dual Pentium III 500 MHz processors, 512 MBytes of RAM, a 9 GByte Quantum Atlas IV SCSI disk, a 100 Mbits/sec Intel EtherExpress Pro fast-ethernet card operating in full-duplex mode, and a 64-bit Myrinet card. We conducted all experiments using fast-ethernet due to some Myrinet instability at the time of experimentation. The nodes are currently using Red Hat 7.1 with kernel 2.4.9 compiled for SMP use. Our I/O configuration included 8 PVFS I/O servers with one I/O server doubling as both a manager and an I/O server. PVFS files were striped with a stripe size of 16 KBytes. MPICH 1.2.4 was used in all our testing using hints for list I/O, data sieving and collective operations. ROMIO was compiled with PVFS version 1.5.6-pre1. All ROMIO data sieving operations and collective operations were performed using a 4 MByte buffer. All results are the average of three runs.

4.2 Tile Reader Benchmark

Tiled visualization code is used to study the effectiveness of commodity based graphics systems in creating parallel and distributed visualization tools. The amount of detail in current visualization methods requires more than a single desktop monitor can resolve. Using two-dimensional displays to visualize large datasets or real-time simulation is important for high performance applications. Our version of the tiled visualization code, the tile reader benchmark, uses multiple compute nodes, with each compute node taking high-resolution display frames and reading only the visualization data necessary for its own display. We use six compute nodes for our testing, which mimics the display size of the full application. The six compute nodes are arranged in the 3 x 2 display shown in Figure 4, each with a resolution of 1024 x 768 with 24-bit color. In order to hide the merging of display edges, there is a 270-pixel horizontal overlap and a 128-pixel vertical overlap. Each frame has a file size of about 10.2 MBytes. A set of 100 frames is read for a total of 1.02 GBytes.

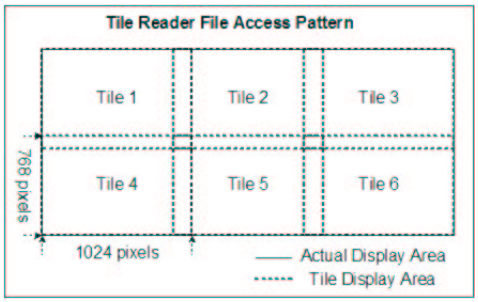

Figure 4: Tile reader file access pattern. Each processor is in charge of reading the data from a display file into its own local display, also known as a tile. This results in a noncontiguous file access pattern.

Table 1 provides a summary of the I/O characteristics of this benchmark for each method of access. We expect ROMIO list I/O to perform best with this access pattern because the noncontiguous file regions are large and there are only 768 noncontiguous file regions. This means that 768 / 128 = 6 I/O requests per processor, which is not too burdensome. ROMIO POSIX I/O will have to make all 768 I/O requests per processor. The data layout is sparse enough to lessen the performance improvements of data sieving in ROMIO data sieving I/O or ROMIO collective I/O. Data sieving ROMIO I/O will perform marginally since only 2.25 MBytes of 5.56 MBytes of the data accessed per processor is useful. Collective ROMIO I/O will be able to use data sieving effectively since no data will be wasted. However the overhead of the second phase of redistribution will make it slower than ROMIO data sieving I/O since it must pass 4.5 MBytes of data to other processors.

We can see in Figure 5 that ROMIO list I/O outperforms the other ROMIO I/O methods in both the uncached and cached read cases. All of the methods perform better in the cached case due to getting data from memory on the I/O servers instead of the disk. While ROMIO collective I/O views the file access pattern as contiguous from an aggregate standpoint, the file regions are too large to enable it to overcome the overhead of reading data once from file and then sending the data to the requesting nodes. In fact, the overhead of file redistribution causes it to fall behind ROMIO data sieving I/O. The overhead of 768 I/O calls to the file system caused ROMIO POSIX I/O to lag far behind the other implementations. Other noncontiguous reading benchmarks perform with similar higher performance trends in cached read cases versus uncached read cases, so we only focus on uncached noncontiguous read performance in the other noncontiguous read tests.

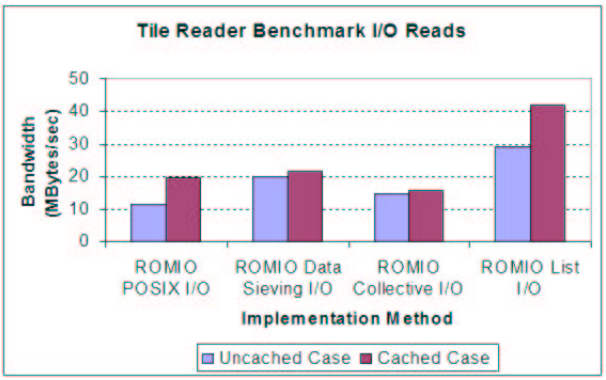

Figure 5: Tile reader benchmark results.

Table 2: I/O Characteristics of the ROMIO Three-Dimensional Block Test

	Desired data per client	Data accessed per client	# of I/O ops per client	Resent data per client
8 Clients				
POSIX I/O	103 MB	103 MB	90,000	—
Data Sieving I/O	103 MB	412 MB	103	—
Collective I/O	103 MB	103 MB	26	77.2 MB
List I/O	103 MB	103 MB	704	—
27 Clients				
POSIX I/O	30.5 MB	30.5 MB	40,000	—
Data Sieving I/O	30.5 MB	274.7 MB	69	—
Collective I/O	30.5 MB	30.5 MB	8	27.1 MB
List I/O	30.5 MB	30.5 MB	313	—
64 Clients				
POSIX I/O	12.9 MB	12.9 MB	22,500	—
Data Sieving I/O	12.9 MB	206.0 MB	52	—
Collective I/O	12.9 MB	12.9 MB	4	12.1 MB
List I/O	12.9 MB	12.9 MB	176	—

4.3 ROMIO Three-Dimensional Block Test

The ROMIO test suite consists of a number of correctness and performance tests. We chose the coll_perf.c test from this suite to compare our methods of noncontiguous data access. The coll_perf.c test measures the I/O bandwidth for both reading and writing to a file with a file access pattern of a three-dimensional block-distributed array. The three-dimensional array, shown graphically in Figure 6, has dimensions 600 x 600 x 600 with an element size of an integer (4 bytes).

Table 2 summarizes the I/O characteristics of this test for the four I/O methods and three numbers of processes. Due to the three-dimensional block access pattern, we expected that increasing the number of processors would have a large effect on the performance. For example when we used 8 processors, data sieving operations would waste 3/4 of the data accessed, roughly 309 Mbytes. When we used 64 processors, data sieving operations would waste 15/16 of the data accessed, roughly 193.1 MBytes. When considering ROMIO collective I/O, no file data would be wasted, but the redistribution size in both the read and write cases would be the same as the wasted file data in ROMIO data sieving I/O. When using 8 or 64 processors, 3/4 or 15/16, respectively, of the data accessed by I/O processors would be redistributed to other processors. ROMIO POSIX I/O will have to face 90,000 accesses per processor with 8 processors, 40,000 access per processor with 27 processors, and 22,500 access with 64 processors. ROMIO list I/O faces a reduced number of accesses versus ROMIO POSIX I/O, but must contend with many more I/O operations per client versus data sieving methods.

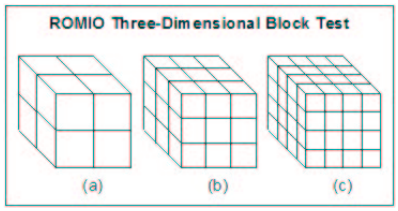

Figure 6: Three-dimensional block test access pattern. The access pattern for 8, 27, and 64 processors is shown in (a), (b), and (c) respectively.

Table 3: I/O Characteristics of the FLASH I/O Simulation (n is the # of clients)

	Desired data per client	Data accessed per client	# of I/O ops per client	Resent data per client
POSIX I/O	7.50 MB	7.50 MB	983,040	—
Data Sieving I/O	—	—	—	—
Collective I/O	7.50 MB	7.50 MB	2	7.5 MB * $\frac{n-1}{n}$
List I/O	7.50 MB	7.50 MB	7,680	—

Figure 7 shows the results of the three-dimensional block test. In the write case, we see ROMIO list I/O take a big lead over the other methods and then drop significantly with 27 processors and 64 processes. We can attribute this slowdown to a smaller contiguous file region size and an increased number of system-wide I/O requests to the I/O servers. ROMIO POSIX I/O performs very poorly due to even more I/O requests than ROMIO list I/O in all cases. ROMIO collective I/O sees some gains in this test with more processors since it performs large contigous writes with the assigned I/O processors instead of small noncontiguous writes like the other methods. In the read case, ROMIO list I/O results improve from 8 to 27 processors due to having more clients outweighing the effect of having smaller accesses. However, at 64 processors, the overhead of increased I/O requests and smaller file regions has lessened performance. ROMIO POSIX I/O performs worse with an increased number of processors due to 128 times more I/O requests than ROMIO list I/O. ROMIO data sieving I/O also performs worse with more processors since it accesses 206 MBytes per client while using only 12.9 MBytes. ROMIO collective I/O suffers from the heavy redistribution cost.

4.4 FLASH I/O Simulation

The FLASH code is an adaptive mesh refinement application that solves fully compressible, reactive hydrodynamic equations, developed mainly for the study of nuclear flashes on neutron stars and white dwarfs [9]. The I/O performance for FLASH determines how often checkpointing may be performed, so I/O performance is critical. The actual FLASH code uses HDF5 for writing checkpoints, but the organization of variables in the file is the same in our simulation. The element data in every block on every processor is written to file by using noncontiguous

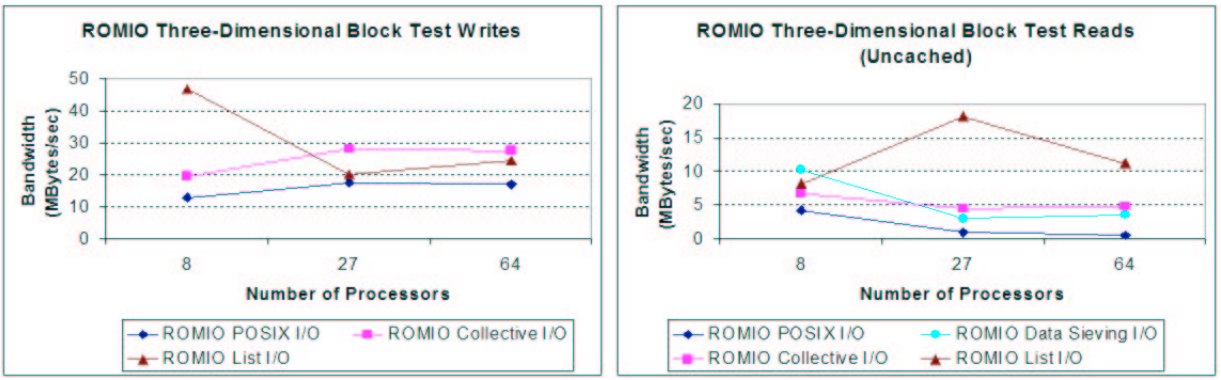

Figure 7: Three-dimensional block test results.

MPI Datatypes. The access pattern of the FLASH code is noncontiguous both in memory and in file, making it a challenging application for parallel I/O systems. The FLASH memory datatype, viewable in Figure 8, consists of 80 FLASH three-dimensional blocks, or cells in the refined mesh, on each processor. Every block contains an inner data block surrounded by guard cells. Each of these data elements has 24 variables associated with it. Every processor writes these blocks to a file in a manner such that the file appears as the data for variable 0, then the data for variable 1, all the way up to variable 23 as shown in Figure 9. Within each variable in file, there exist 80 blocks, each of these blocks containing all the FLASH blocks from every processor. Since every processor writes 80 FLASH blocks to file, as we increase the number of clients, the dataset size increases linearly as well. Every processor adds 7 MBytes to the file, so the dataset ranges between 14 MBytes (2 clients) to 448 MBytes (64 clients).

Table 3 summarizes the I/O characteristics of this benchmark for the access methods tested, with *n* referring to the number of compute processors (or clients). Note that because this is a write benchmark and PVFS does not have file locking, the data sieving method was not tested. In our testing we vary the number of clients from 2 to 64. Each contiguous memory region is the size of a double (8 bytes). In file, however, the contiguous regions are (8 x-elements)*(8 y-elements)*(8 z-elements)*(sizeofdouble) = 4096 bytes. The FLASH I/O code is worst for the ROMIO POSIX I/O approach since the noncontiguous file region access pattern is sparse. The number of I/O requests for ROMIO default I/O = (80 blocks)*(8 x-elements)*(8 y-elements)*(8 z-elements)*(24 variables) = 983,040 I/O calls per processor. Our implementation of ROMIO list I/O can do a little better since ROMIO list I/O can describe noncontiguous file regions in a single I/O calls. However, since our maximum limit of file regions was set at 128 for this test, 983,040 / 128 = 7,680 I/O calls per processor are still required. The FLASH I/O benchmark presents a good opportunity to use ROMIO collective I/O since the aggregate view of the file is actually a contiguous region of file. ROMIO collective I/O only needs to perform I/O calls = (Aggregate data size)/((number of procesors)*(Buffer size)) = (7 MBytes * N procs)/(N procs)*(4 Mbytes) = 7/4 rounded up to 2 I/O calls per processor. While, $\frac{n-1}{n}$ of the data must be redistributed, the savings in I/O calls is significant enough to overcome the redistribution overhead. We expect ROMIO collective I/O to perform best in this benchmark.

Figure 10 shows that ROMIO collective I/O works more efficiently than the other ROMIO I/O methods for numerous noncontiguous file regions that appear contiguous from a global standpoint. ROMIO POSIX I/O suffers from the immense overhead of 983,040 I/O calls per processor while list I/O does roughly two magnitudes better due to only having 7,680 I/O calls per processor. As we increase the number of processors, the dataset size increases. We see that both ROMIO POSIX I/O and ROMIO list I/O increase bandwidth with more processors, but ROMIO collective I/O starts to fall with larger dataset sizes between 16 to 32 nodes. For the two processor case of 14 MBytes, in ROMIO collective I/O, 4 contiguous writes could cover the entire file, but at 448 MBytes, 112 contiguous writes are necessary. Even though it is only 2 contiguous writes per processor in all cases, redistribution becomes more and more expensive as we increase the aggregate data size. For example, with 64 processors, only 1/64 of the data read from the file system is used by the processor doing I/O, the other 63/64, 12.1 MBytes is sent to other processors.

5 Related Work

Several vendors of MPI-IO implement MPI read and write calls in various ways to improve performance. For example, the MPI-IO implementation on GPFS uses the data-shipping technique and controlled prefetching to reduce the number of file I/O operations per MPI-IO read/write calls [14]. When using controlled prefetching, GPFS analyzes the predefined pattern of I/O requests to find the lower and upper bounds of the byte range, which is essentially a data sieving operation, since the file is read into memory at an I/O agent and then data movement is performed. While this strategy can result in good performance for certain data access patterns, there are several problems with prefetching. For file accesses that are logically distant (for example an access that spans an entire file), byte range locking grows exceptionally large and slow, even if the data movement consists only of a byte at the beginning of the lock and a byte at the end of the lock. Also, in the write case, file locking slows performance considerably in many cases. The user must choose between dealing with these problems or turning off prefetching and having I/O agents send multiple requests for noncontiguous data.

The High Performance Storage System (HPSS) is commercial software for hierarchical storage management and services in large-scale storage [20]. HPSS has support

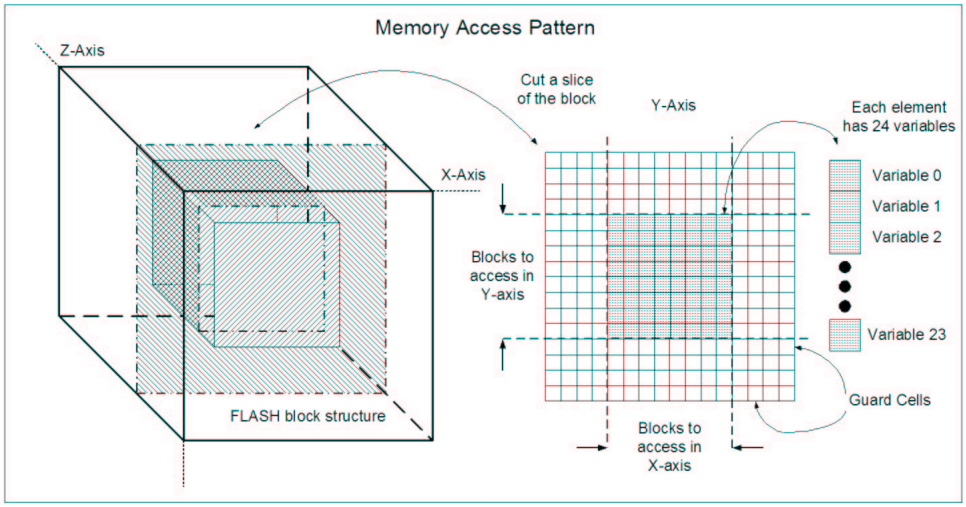

Figure 8: FLASH memory datatype. Each computing processor contains 80 blocks, so as we scale up the number of computing processors, we linearly increase the dataset size.

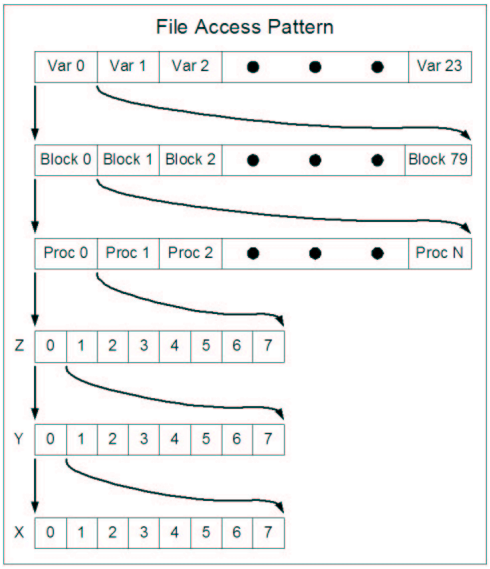

Figure 9: FLASH file datatype. This figure describes the hierarchy of the file datatype. At the highest level of the hierarchy, variables are contiguous. Within every variable, there are all the FLASH blocks from all the processors.

for MPI-IO and implements MPI-IO read/write calls using several optimizations including asynchronous operation, grouping accesses on the same storage, grouping accesses on the same processor and coalescing of small accesses [11]. Unfortunately, since HPSS is commercial software, little documentation is available on the implementation.

6 Conclusions and Future Work

ROMIO list I/O is a significant implementation for noncontiguous data access. We have seen the performance improvements in both the tile reader benchmark as well as the noncontiguous ROMIO benchmark. In many noncontiguous cases, ROMIO list I/O can outperform current ROMIO methods, but there are cases, such as the FLASH I/O benchmark in Section 4.4 where the number of noncontiguous file regions grows too large for ROMIO list I/O to reduce linearly.

While the list I/O interface is a major step in support for noncontiguous accesses, it is not optimal. Particularly in the case of MPI-IO, noncontiguous accesses often have regular patterns in the file, and these patterns are described concisely in the datatypes passed to the MPI I/O call. The list I/O interface, as described here, loses these regular patterns, instead flattening them into potentially large lists of contiguous regions. One area of future study is in how these descriptions of regular patterns can be retained and passed directly to the parallel file system. This capability can have a significant impact on the size of the I/O request, which can be extremely important in cases where many noncontiguous regions each of a very small size are being accessed.

Acknowledgements

This work was funded in part by National Aeronautics and Space Administration grants NAG5-3835 and NGT5-21 and in part 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.

References

[1] Troy Baer. Parallel I/O Experiences on an SGI 750 Cluster. In *CUG Summit 2002*, May 2002.

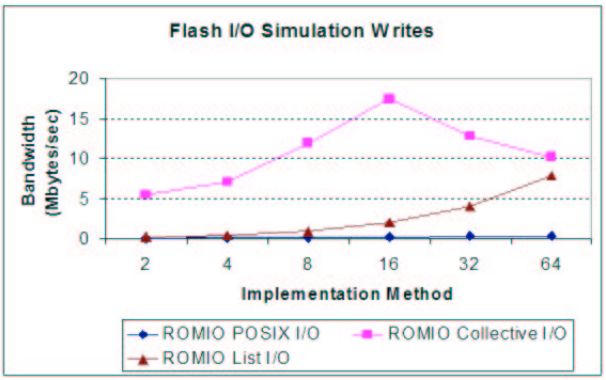

Figure 10: Results of the FLASH I/O benchmark with 2 - 64 processors. Collective I/O performs exceptionally well due to the aggregate contiguous file access pattern. The overhead of exchanging data is minimal compared to the I/O time.

[2] Sandra Johnson Baylor and C. Eric Wu. Parallel I/O workload characteristics using Vesta. In *Proceedings of the IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 16–29, April 1995.

[3] Rajesh Bordawekar, Juan Miguel del Rosario, and Alok Choudhary. Design and evaluation of primitives for parallel I/O. In *Proceedings of Supercomputing '93*, pages 452–461, Portland, OR, 1993. IEEE Computer Society Press.

[4] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, October 2000. USENIX Association.

[5] Chiba City, the Argonne scalable cluster. `http://www.mcs.anl.gov/chiba/`.

[6] Avery Ching, Alok Choudhary, Wei keng Liao, Robert Ross, and William Gropp. Noncontiguous I/O through PVFS. In *Proceedings of the 2002 IEEE International Conference on Cluster Computing*, September 2002.

[7] 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*, San Diego, CA, December 1995. IEEE Computer Society Press.

[8] 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 IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 56–70, Newport Beach, CA, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 31–38.

[9] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, and H. Tufo. FLASH: An adaptive mesh hydrodynamics code for modelling astrophysical thermonuclear flashes. *Astrophysical Journal Suppliment*, 131:273, 2000.

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

[11] 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.

[12] David Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74. USENIX Association, November 1994. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994.

[13] 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.

[14] Jean-Pierre Prost, Richard Treumann, Richard Hedges, Bin Jia, and Alice Koniges. MPI-IO/GPFS, an Optimized Implementation of MPI-IO on top of GPFS. In *Proceedings of SC2001*, November 2001.

[15] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995. IEEE Computer Society Press.

[16] Rajeev Thakur, William Gropp, and Ewing Lusk. An abstract-device interface for implementing portable parallel-I/O interfaces. In *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 180–187, October 1996.

[17] 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*. ACM Press, November 1998.

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

[19] 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.

[20] Richard W. Watson and Robert A. Coyne. The parallel I/O architecture of the high-performance storage system (HPSS). In *Proceedings of the Fourteenth IEEE Symposium on Mass Storage Systems*, pages 27–44. IEEE Computer Society Press, September 1995.