Evaluating Structured I/O Methods for Parallel File Systems

Avery Ching, Alok Choudhary, Wei-keng Liao, Robert Ross, William Gropp

***Abstract*— Modern data-intensive structured datasets constantly undergo manipulation and migration through parallel scientific applications. Directly supporting these time consuming operations is an important step in providing high-performance I/O solutions for modern large-scale applications. High-level interfaces such as HDF5 and Parallel netCDF provide convenient APIs for accessing structured datasets, and the MPI-IO interface also supports efficient access to structured data. Parallel file systems do not traditionally support such structured access from these higher level interfaces.**

In this work we present two contributions. First, we demonstrate an implementation of structured data access support in the context of the Parallel Virtual File System (PVFS). We call this support "datatype I/O" because of its similarity to MPI datatypes. This support is built by using a reusable datatype-processing component from the MPICH2 MPI implementation.

The second contribution of this work is a comparison of I/O characteristics when using modern high-performance noncontiguous I/O methods. We use our I/O characteristics comparison to assess the performance of all the noncontiguous I/O methods using three test applications. We also point to further optimizations that could be leveraged for even more efficient operation.

***Index Terms*— datatype, evaluation, noncontiguous, parallel I/O, PVFS, high performance, MPI-IO, I/O.**

I. Introduction

SCIENTIFIC applications have begun to rely heavily on high-level I/O APIs such as HDF5 [1] and parallel netCDF [2] for their storage needs. These APIs allow scientists to describe their data in meaningful terms to them, as structured, typed data, and to store and retrieve this data in a manner that is portable across all the platforms they might find useful. Because scientists have richer languages with which to describe their data, I/O for an application as a whole can be described in terms of the datatypes and organizations that the scientists are really using, rather than in terms of independent reads or writes of bytes on many processors.

These APIs also allow I/O experts to embed the knowledge of how to efficiently access storage resources in a library that many applications can use. The result is a big win for both groups. Implementors of high-level I/O libraries in turn use MPI-IO as their interface to storage resources. This lower-level interface maps higher-level accesses to file system operations and provides a collection of key optimizations. MPI-IO also understands structured data access, enabling high-level I/O API programmers to describe noncontiguous accesses as single units, just as the scientist did, and to interface to underlying resources through a portable API.

Today's parallel file systems do not, for the most part, support structured or even noncontiguous accesses. Instead they support the POSIX interface, allowing for only contiguous regions to be accessed and modified. This approach severely limits the ability of the MPI-IO layer to succinctly and efficiently perform the accesses that have been described by these higher layers.

A significant step in the direction of efficient structured data access is the *list I/O* interface [3], implemented in the Parallel Virtual File System (PVFS) and supported under MPI-IO [4], [5]. This new interface, when well supported by the parallel file system, allows structured accesses to be described and serves as a solid building block for an MPI-IO implementation. This interface is general, easy to understand, and could be implemented for most file systems. Emerging file systems will likely have such an interface. However, because it does not retain any information on the regularity of access, such as stride information, the access pattern representation of structured accesses can be very large. Building, transmitting, and processing this representation can significantly limit performance when accesses consist of many small regions [6].

In this work we investigate the next logical step in efficient support for structured data access. The approach, *datatype I/O*, provides a mechanism for mapping MPI datatypes (passed to MPI-IO routines) into a type representation understood by the file system. The new representation maintains the concise descriptions possible with MPI type constructors such as `MPI_Type_vector`. This representation is passed over the network to I/O servers, which may process this directly, avoiding the overhead of building lists of I/O regions at the MPI-IO layer, passing these lists over the network as part of the file system request, or processing these lists during I/O.

In addition to introducing this new noncontiguous I/O method, we also present an analytic comparison of the various noncontiguous I/O methods. From these comparisons, we can infer which noncontiguous I/O operations favor particular noncontiguous I/O methods. We describe how to determine the performance characteristics of the various noncontiguous I/O methods for several major I/O characteristic parameters. These I/O characteristics are then used for each benchmark to help understand the performance comparison between differing noncontiguous I/O methods.

In Section II we explain the existing approaches for performing noncontiguous access, including list I/O. In Section III we describe our prototype implementation of datatype I/O, the component on which it is built, how datatype I/O is

Manuscript last revised February 4, 2004.

A. Ching, A. Choudhary, and W. Liao are with Northwestern University.
R. Ross and and W. Gropp are with Argonne National Laboratory.

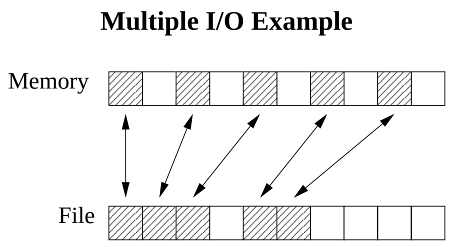

Fig. 1. Example POSIX I/O call. Using traditional POSIX interfaces for this access pattern costs five I/O calls, one per contiguous region.

integrated into the parallel file system, and some limitations of the prototype implementation. In Section IV we present an in-depth analysis of the I/O characteristics of the various noncontiguous I/O methods. In Section V we examine the performance of our prototype using three benchmarks: a tile reading application, a three-dimensional block decomposition I/O kernel, and a simulation of the FLASH I/O checkpoint process. We use the I/O characteristics analysis to aid in understanding the results. In Section VI we discuss related efforts and future directions for this work.

II. Current Noncontiguous I/O Approaches

Noncontiguous data is simply data that resides in different areas, with gaps between them. Noncontiguous I/O refers to an I/O operation in which data in memory, in file, or in both is noncontiguous. Several approaches have been implemented for supporting noncontiguous I/O access. The first, and most naive, is the approach we call *POSIX I/O*.

A. POSIX I/O

Most parallel file systems implement the POSIX I/O interface [7]. This interface provides the capability to perform contiguous data access only. To support noncontiguous access with POSIX I/O, one must break the noncontiguous I/O into a sequence of contiguous I/O operations. This approach to noncontiguous I/O access requires significant overhead in the number of I/O requests that must be processed by the underlying file system. As shown in Figure 1, even simple noncontiguous access patterns can result in numerous contiguous I/O operations. Because operations in parallel file systems often require data movement over a network, latency for I/O operations can be high. For this reason performing many small I/O operations to service a noncontiguous access is very inefficient. Fortunately for users of these file systems, two important optimizations have been devised for more efficiently performing noncontiguous I/O using only POSIX I/O calls: data sieving I/O and two-phase I/O.

B. Data Sieving I/O

The *data sieving* method reduces the number of I/O operations while simultaneously increasing I/O operation sizes [8]. Increasing the size of an I/O operation allows the hard drive to provide higher I/O bandwidth. When data sieving is used, a large region encompassing all the data in the file is accessed

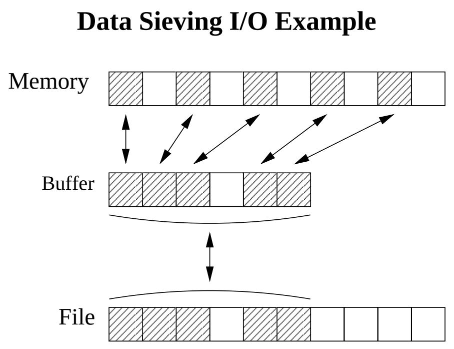

Fig. 2. Example data sieving I/O call. By first reading a large contiguous file region into a buffer, data movement is subsequently performed between memory and the buffer.

with a minimum number of POSIX I/O operations. For read operations, a large contiguous data region containing desired data is first read into a temporary buffer, and the desired data is then extracted into the user's buffer as shown in Figure 2. For write operations, a read-modify-write sequence is performed. A large contiguous region is read into a temporary buffer, new data is placed into the appropriate positions in this buffer, and the buffer is then written back to storage. In order to ensure consistency during concurrent operations, a lock must be held on the region to be modified during the read-modify-write process. Depending on the file locking implementation, the locking overhead can be mild or serious. File locking implementations will serialize I/O access if the noncontiguous data requested by multiple processors is either interleaved or overlapping.

This approach is efficient when the desired noncontiguous regions exhibit good spatial locality (i.e., are close together). When data is more dispersed; however, data sieving accesses a great deal of additional data in order to perform the operation and, at some point, becomes less effective than simply using a sequence of POSIX I/O calls.

C. Two-Phase I/O

When used to their fullest, interfaces such as MPI-IO give a great deal of information about how the application as a whole is accessing storage. One example of this is the collective I/O calls that are part of the MPI-IO API. By making collective I/O calls, applications tell the MPI-IO library not only that each process is performing I/O but also that these I/O operations are part of a larger whole. This information provides additional opportunities for optimization over application processes performing independent operations.

Two-phase I/O, developed by Thakur et al., is one example of a collective I/O optimization [9]. The two-phase method builds on POSIX I/O and data sieving I/O. The two-phase method identifies a subset of the processes that will actually perform I/O; these processes are known as *aggregators*. Each aggregator is responsible for I/O to a specific portion of the file; the implementation in ROMIO calculates these regions dynamically based on the size and location of the accesses in the collective operation.

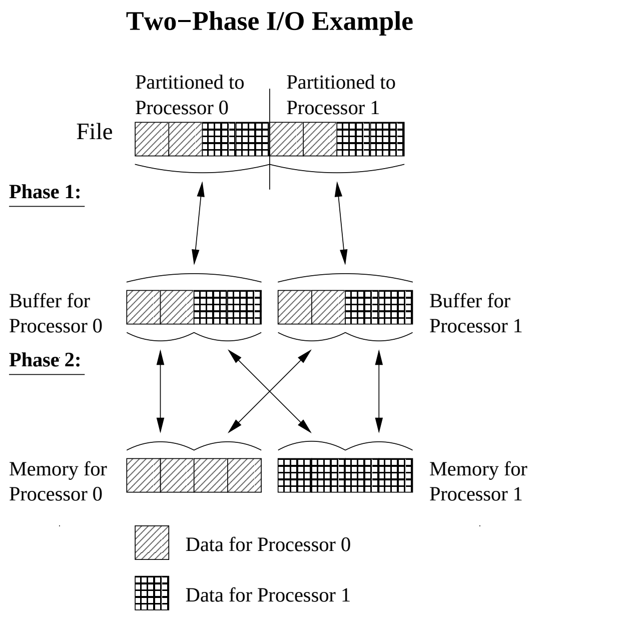

Fig. 3. Example two-phase I/O call. Interleaved file access patterns can be effectively accessed in larger file I/O operations with the two-phase I/O method.

Read operations using the two-phase method are performed as shown in Figure 3. First, aggregators read a contiguous region containing desired data from storage and put this data in a temporary buffer. Next, data is redistributed from these temporary buffers to the final destination processes. Write operations are performed in a similar manner. First, data is gathered from all processes into temporary buffers on aggregators. Next, this data is written back to storage using POSIX I/O operations. An approach similar to data sieving is used to optimize this write back to storage in the case where there are still gaps in the data. Alternatively, other noncontiguous access methods, such those described in upcoming sections, can be leveraged for further optimization.

Two-phase I/O has a distinct advantage over data sieving alone in that it is far more likely to see dense regions of desired data because of combining the regions of many processes. Hence, the reads and writes in two-phase I/O are more efficient than data sieving in many cases. However, two-phase I/O also relies on the MPI implementation providing high-performance data movement. If the interprocess communication is not significantly faster than the aggregate I/O bandwidth in the system, the overhead of the additional data movement in two-phase I/O is likely to prevent it from outperforming the direct access optimizations (data sieving I/O, list I/O, and datatype I/O).

D. List I/O

List I/O [3] is an enhanced parallel file system interface designed to support accesses that are noncontiguous in both memory and file (see prototypes in Figure 5). With this interface an MPI-IO implementation can *flatten* the memory and file datatypes (convert them into lists of contiguous regions) and then describe an MPI-IO operation with a single list I/O call as shown in Figure 4. Given an efficient implementation in the parallel file system, this interface can significantly boost performance. In previous works we discussed the implementation of list I/O in PVFS and support for list I/O [4] under the ROMIO MPI-IO implementation [5]. Since the list I/O method is general enough for simple noncontiguous I/O, there is a good possibility it will be adopted by current and future file systems.

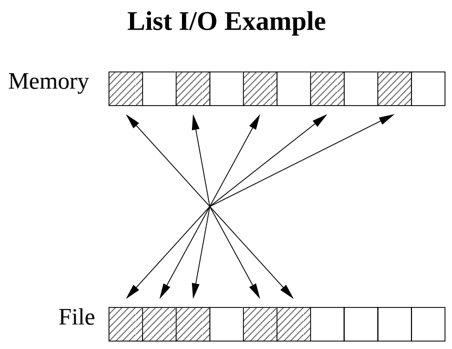

Fig. 4. Example list I/O call. Only a single I/O request is necessary to handle this noncontiguous access because of more descriptive I/O requests.

The major drawbacks of list I/O are the creation and processing of these large lists and the transmission of these lists from client to server inside the parallel file system layer. Additionally, given that we want to bound the size of I/O requests within the file system, only a fixed number of regions can be described in one request. Thus, while list I/O does significantly reduce the number of I/O operations versus POSIX I/O (in our implementation by a factor of 64), a linear relationship still exists between the number of noncontiguous file regions and the number of I/O operations (within the file system layer). Hence, while list I/O is an important addition to the optimizations available under MPI-IO, it does not replace two-phase or data sieving; rather, it augments them.

III. Datatype I/O

Datatype I/O is an effort to address the deficiencies seen in the list I/O interface when faced with accesses that are made up of many small regions, particularly ones that exhibit some degree of regularity. Datatype I/O borrows from the datatype concept that has proven invaluable for both message passing and I/O in MPI applications. The constructors used in MPI datatypes allow for concise descriptions of the regular, noncontiguous data patterns seen in many scientific applications, such as extracting a row from a two-dimensional dataset.

The datatype I/O interface, shown in Figure 6, replaces the lists of I/O regions seen in the list I/O interface with an address, count, and datatype for memory and a displacement, datatype, and offset into the datatype for file. These correspond directly to the address, count, datatype, and offset into the file view passed into an MPI-IO call and the displacement and file view datatype previously defined for the file. The datatype I/O interface is not meant to be used by application programmers; it is an interface specifically for use by I/O library developers. Helper routines are used to convert MPI datatypes into the

```
int listio_read(int fd, int mem_list_count, void *mem_offsets[],
                int mem_lengths[], int file_list_count, int file_offsets[],
                int file_lengths[])

int listio_write(int fd, int mem_list_count, void *mem_offsets[],
                 int mem_lengths[], int file_list_count, int file_offsets[],
                 int file_lengths[])
```

Fig. 5. List I/O prototypes

```
int dtype_read(int fd, void  *mem_addr, int mem_dtype_count, dtype *mem_dtype,
               int file_dtype_disp, int offset_into_dtype, dtype *file_dtype)

int dtype_write(int fd, void  *mem_addr, int mem_dtype_count, dtype *mem_dtype,
                int file_dtype_disp, int offset_into_dtype, dtype *file_dtype)
```

Fig. 6. Datatype I/O prototypes

format used by the datatype I/O functions. A full-featured implementation of datatype I/O would

- maintain a concise datatype representation locally and avoid datatype flattening,
- use this concise datatype representation when describing accesses, and
- service accesses using a system that processes this representation directly.

Our prototype implementation of datatype I/O was written as an extension to the Parallel Virtual File System. The ROMIO MPI-IO implementation was likewise modified to use datatype I/O calls for PVFS file system operations.

We emphasize that while we present this work in the context of MPI-IO and MPI datatypes, nothing precludes our using the same approach to directly describe datatypes from other APIs, such as HDF5 hyperslabs; in fact, because HDF5 uses MPI-IO it can benefit from this improvement without code changes.

A. The Parallel Virtual File System and ROMIO MPI-IO Implementations

The Parallel Virtual File System is a parallel file system for commodity Linux clusters [10]. It provides both a clusterwide consistent name space and user-defined file striping. PVFS is a client-server system consisting of clients, a metadata server, and I/O servers. Clients retrieve a list of the I/O servers that contain the file data from the metadata server at file open time. Subsequent reading or writing is processed directly by the I/O servers without manager interaction.

The approach that PVFS uses for processing requests is detailed in [11]. In short, PVFS builds a data structure called a *job* on each client and server for every client/server pair involved in an I/O operation. This structure points to a list of *accesses*, which are contiguous regions in memory (on a client) or in file (on a server) that must be moved across the network. This is essentially the flattened representation of the datatype being used to move data. While this is not ideal from a processing overhead standpoint, we will retain this representation in our tests; it would be time consuming to reimplement this core component of PVFS.

ROMIO is the MPI-IO implementation developed at Argonne National Laboratory [12]. It builds on the MPI-1 message-passing operations and supports many underlying file systems through the use of an abstract interface for I/O (ADIO). ADIO allows the use of file system specific optimizations such as the list I/O and datatype I/O interfaces described here. Additionally, ROMIO implements the data sieving and two-phase optimizations described in Sections II-B and II-C. It also implements a datatype flattening system that is used to support list I/O for PVFS.

B. Datatype I/O Implementation in PVFS and ROMIO

Our datatype I/O prototype builds on the datatype processing component in MPICH2 [13]. Three key characteristics of this implementation make it ideal for reuse in this role:

- Simplified type representation (over MPI datatypes)
- Support for partial processing of datatypes
- Separation of type parsing from action to perform on data

Types are described by combining a concise set of descriptors called *dataloops*. Dataloops can be of five types: contig, vector, blockindexed, indexed, and struct [14]. These five types capture the maximum amount of regularity possible, keeping the representation concise. At the same time these are sufficient to describe the entire range of MPI datatypes. Simplifying the set of descriptors aids greatly in implementing support for fast datatype processing because it reduces the number of cases that the processing code must handle. The type extent is retained in this representation (a general concept) while the MPI-specific LB and UB values are eliminated. This simplification has the added benefit of allowing resized type processing with no additional overhead in our representation. We use dataloops as the native representation of types in our PVFS implementation.

The MPICH2 datatype component provides the functions necessary to process dataloop representations [13]. We provide functions to convert MPI datatypes into dataloops and functions that are called during processing to create the offset-length pairs we need to build the PVFS job and access structures. Additionally we provide functionality for shipping dataloops as part of I/O requests. In our prototype, MPI datatypes are converted to dataloops by a recursive process built by using the functions `MPI_Type_get_envelope` and `MPI_Type_get_contents`. By using these MPI functions,

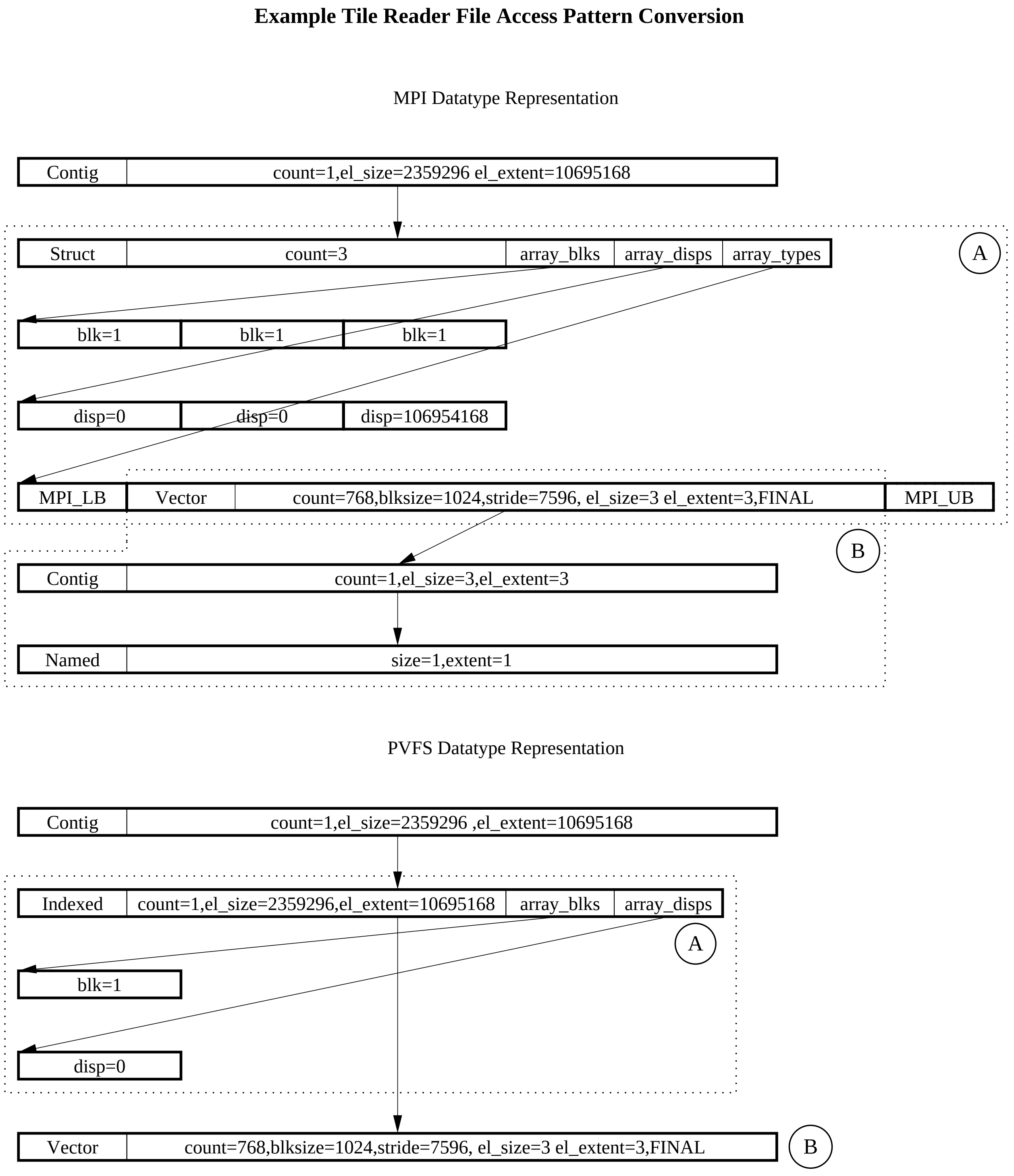
Example Tile Reader File Access Pattern Conversion
MPI Datatype Representation
Contig
count=1,el_size=2359296 el_extent=10695168
Struct
count=3
array_blks
array_disps
array_types
A
blk=1
blk=1
blk=1
disp=0
disp=0
disp=106954168
MPI_LB
Vector
count=768,blksize=1024,stride=7596, el_size=3 el_extent=3,FINAL
MPI_UB
B
Contig
count=1,el_size=3,el_extent=3
Named
size=1,extent=1
PVFS Datatype Representation
Contig
count=1,el_size=2359296 ,el_extent=10695168
Indexed
count=1,el_size=2359296,el_extent=10695168
array_blks
array_disps
A
blk=1
disp=0
Vector
count=768,blksize=1024,stride=7596, el_size=3 el_extent=3,FINAL
B

Fig. 7. Example tile reader file access pattern conversion. (A) shows how we convert a struct datatype into an indexed dataloop for performance optimization. This conversion eliminates the need for the MPI_LB and MPI_UB dataloops, making the dataloop representation smaller. (B) is an example of loop fusion in which we can merge datatypes into a single dataloop. The contig and named dataloops can be sufficiently described by the vector dataloop above them, eliminating the need for them.

we can ensure the portability of our datatype I/O method across different MPI implementations.

An optimization our conversion process employs is the reduction of the size of the access pattern representation. By using loop fusion as well as a struct-to-indexed datatype conversion, we can reduce the amount of data transferred over the network. Loop fusion is the conversion of excessively created dataloops into more concise dataloops, reducing the size of the overall dataloop structure while maintaining the desired access pattern. The struct-to-indexed datatype conversion we employ converts a struct dataloop (which may have multiple underlying dataloops) into an indexed dataloop, which has only one underlying dataloop), generally resulting in a smaller access pattern representation. Both of these optimizations are visualized in an example file access pattern conversion from the tile reader test in Section V-B in Figure 7. The resulting dataloop representation is passed into the datatype I/O calls and from there sent to the relevant I/O servers. The dataloops are converted into the job and access structures on servers and clients side to create the traditional PVFS job and access structures. Figure 8 outlines this process. PVFS-specific functions for creating these offset-length pairs are passed to the dataloop processing component. These functions are written to efficiently convert contiguous, vector, and indexed dataloops into offset-length pairs, and they include optimizations to coalesce adjacent regions. The partial processing capabilities of the datatype processing component are used to limit the overhead of storing the intermediate offset-length pairs that are created through dataloop processing.

This is only a partial implementation of the datatype I/O approach; a complete approach would avoid creating of lists of regions on server and client. However, we will show that even without this final capability, our prototype exhibits clear performance benefits over the other approaches.

Because the MPI datatypes are converted into dataloops at every MPI I/O operation, we expect there to be slightly higher overhead in the local portion of servicing these operations in comparison with list I/O. On the other hand, because we are concisely describing these types, we expect to see significantly less time spent moving the I/O description across the network. Future optimizations could cache these dataloop representations on clients and servers to ameliorate this overhead.

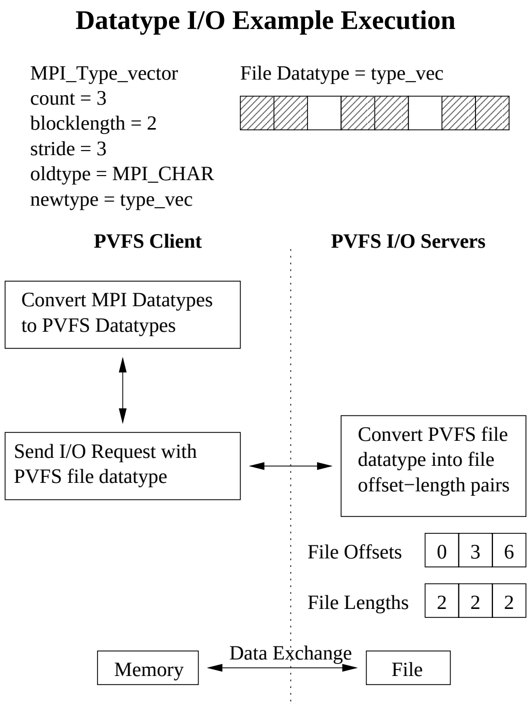

Fig. 8. Example datatype I/O call. Since file datatype are broken into file offset-length pairs at the I/O servers, the number of I/O requests is dramatically reduced for regular access patterns.

IV. Noncontiguous I/O Comparison

The noncontiguous I/O methods described in the previous section have several important performance characteristics that will be discussed in this section. We will focus on the characteristics that have the greatest performance impact. In Table I we show how the major I/O characteristics compare.

We used the following variables to define this comparison: n = number of bytes, *file ext* = the extent of the noncontiguous file regions (from first byte of the first file region to the last byte of the last file region) for a single client, *agg file ext* = the extent of the noncontiguous file regions (from first byte of the first file region to the last byte of the last file region) for the aggregate I/O access pattern, *ds buffer* = the size of the data sieving I/O buffer, *tp buffer* = the size of the two-phase I/O buffer, *fr count* = number of file regions, *fr per access* = the maximum number of file regions a list I/O operation can support before breaking up into multiple list I/O operations. *# of I/O aggs* = the number of I/O aggregators *ap depend* = access pattern dependent, and *fr size* = the size of a file regions (we assume same size file regions for simplicity).

We note that in the performance characteristics of the data sieving I/O and the two-phase I/O there a small approximation is made with respect to the filing of the intermediary buffer. The data sieving I/O buffer is the client's memory region used to do the initial access of data from the I/O system. The two-phase I/O buffer is the memory of the I/O aggregators that access data from the I/O system. Any holes between noncontiguous file regions that are past the last file region in the buffer can be ignored when refilling the buffer in the next iteration. The frequency of this hole being removed from being accessed decreases as the buffer is larger. Therefore, to simplify our calculations, we are essentially approximating an infinitely large intermediary buffer.

A. Desired Data per Client

Each client needs a certain amount of data before it can return completion of its I/O operation. The desired data per client is this final data size. The desired data per client will be the same for each of the noncontiguous I/O methods for a given I/O access pattern.

B. Data Accessed per Client

The client must acquire data from the I/O storage system in order to service its request. For POSIX I/O, list I/O, and

	Desired Data per Client	Data Accessed per Client	# of I/O Ops per Client	Resent Data per Client	File Region Size Accessed
POSIX I/O	*n*	*n*	≥*fr count*	—	≤*fr size*
Data Sieving I/O	*n*	*file ext*	$\lceil \frac{file\ ext}{ds\ buffer} \rceil$	—	≤*ds buffer*
Two-Phase I/O	*n*	$\frac{agg\ file\ ext}{\#\ of\ I/O\ aggs}$	$\lceil \frac{agg\ file\ ext}{\#\ of\ I/O\ aggs * tp\ buffer} \rceil$	*ap depend*	≤*tp buffer*
List I/O	*n*	*n*	$\lceil \frac{fr\ count}{fr\ per\ access} \rceil$	—	*fr size*
Datatype I/O	*n*	*n*	1	—	*fr size*

TABLE I
I/O CHARACTERISTICS COMPARISON

datatype I/O, the data accessed per client is equal to the desired data per client. For data sieving I/O, the data accessed per client is the sum of the desired data per client and the holes between the file regions. For two-phase I/O, the data accessed per client is the amount of data that is actually exchanged between I/O aggregators (all compute nodes in our tests) and the I/O servers; it does not include the cost of the data that is resent between I/O aggregators and compute nodes. The implementation of two-phase I/O leads to the actual amount of data accessed per client being based on *agg file ext* divided by the number of I/O aggregators.

C. I/O Operations per Client

For POSIX I/O, the number of I/O operations per client depends on the number of noncontiguous file regions. For every file region, POSIX I/O requires at least one I/O operation. A mismatch between memory regions and file regions can cause multiple contiguous I/O operations to service a single file region. Therefore, in POSIX I/O, the number of I/O operations per client is greater than or equal to the number of file regions in the access pattern. For data sieving I/O, the number of I/O operations is approximately the ceiling of the *file ext* divided by size of the data sieving buffer. For two-phase I/O, the number of I/O operations per I/O aggregator is equal to the ceiling of the data accessed per client divided by the size of the two-phase buffer. For list I/O, because the implementation of list I/O limits the number of file offset-length pairs that can be passed with a single I/O operation, the number of I/O operations per client is equal to the total number of file regions divided by the number of file offset-length pairs allowed per I/O operation. For datatype I/O, the number of I/O operations is always one per MPI-IO operation because the datatype I/O request includes a derived datatype that can correspond to any MPI-IO datatype.

D. Resent Data per Client

Only two-phase I/O actually resends data between clients. These transfers are generally not so costly as the data accessed per client because they are from the memory of the I/O aggregator to the memory of the client; hence it is a memory-memory network transfer instead of a disk-memory network transfer. Clients using data sieving I/O access regions of the data sieving buffer in the same manner, but this is done locally on the same processor, so we do not include this overhead in this section. We also do not define a formula for this variable because it is highly dependent on the access pattern. For example, if each aggregator is accessing data in its own aggregator file region, then there no data is resent. If each aggregator is accessing data in another aggregators file region, then all the data that is accessed is resent. The amount of resent data per client is best calculated based on the access pattern. In Section V, we will determine the resent data size of the I/O access patterns from each of the benchmarks.

E. File Region Size Accessed

For list I/O and datatype I/O, the file region size is the same as specified in the access pattern. For POSIX I/O, the file region size is also the same as specified in the access pattern except when the memory regions and file regions do not align. If the regions do not align, the I/O operations can be for smaller sizes than the access pattern file region. For data sieving I/O, the file region size is equal to the minimum of the data sieving buffer size or the extent of the remaining file regions. For two-phase I/O, the file region size is equal to the minimum of the two-phase I/O buffer or the extent of the remaining aggregator region extent. When we use this I/O characteristic in Section V, for simplicity we fill in the most commonly used file region size accessed.

V. PERFORMANCE EVALUATION

To evaluate the performance of the datatype I/O optimization against the other noncontiguous I/O methods, we ran a series of noncontiguous MPI-IO tests, including a tile reader benchmark, a three-dimensional block access test, and the FLASH I/O simulation. For each test we provide a table summarizing the I/O characteristics of the access pattern using the metrics from Section IV for each of the noncontiguous I/O methods.

A. Benchmark Configuration

Our results were gathered on Chiba City at Argonne National Laboratory [15]. Chiba City has 256 nodes available with dual Pentium III processors, 512 MBytes of RAM, a single 9 GByte Quantum Atlas IV SCSI drive, and a 100 Bits/sec Intel EtherExpress Pro fast-Ethernet card operating in full-duplex mode. Each node uses RedHat 7.3 with kernel 2.4.21-rc1 compiled for SMP use. Our PVFS server configuration for all test cases included 16 I/O servers (one also doubled as a metadata server). PVFS files were created with a 64 KByte strip size (1 MByte stripes across all servers). In the tile reader tests we allocate one process per node because so few nodes are involved. In the other two cases we allocate two processes per node.

Our prototype was built using the ROMIO version 1.2.4 and PVFS version 1.5.5. All data sieving I/O and two-phase I/O operations were conducted with a 4 Mbyte buffer size (ROMIO default size). Our results are the average of three test runs. All write test times include the time for the MPI_File_sync() command to complete besides the normal write I/O time. All read tests are uncached. We added these constraints to our testing environment to avoid simply testing network bandwidth. Instead, we want to examine the performance of our file system optimizations in conjunction with the storage system hard drive performance.

All read benchmarks are conducted with POSIX I/O, data sieving I/O, two-phase I/O, list I/O, and datatype I/O. All write benchmarks are conducted with POSIX I/O, two-phase I/O, list I/O, and datatype I/O. ROMIO can support write operations with data sieving I/O only if file locking is supported by the underlying file system. Since PVFS does not support file locking, we cannot perform data sieving writes on PVFS. We note, however, that for file systems that do allow file locking, data sieving performance for writes will have worse performance than data sieving reads for the same access pattern. Data sieving writes perform the same data movement from file system into the data sieving buffer at the client and then data movement from data sieving buffer to memory buffer, but they also have to write this data back, thereby doubling the network data transfer of a data sieving read. Also, locking the modified regions can cause serialization of I/O requests for overlapping requests, another serious overhead. On the other hand, the MPI-IO consistency semantics do allow us to perform a read-modify-write during collective I/O. Thus, an approach similar to data sieving is used in the two-phase I/O write case.

B. Tile Reader Benchmark

Often the amount of detail in scientific application visualization exceeds the display capabilities of a single desktop monitor. In these cases an array of displays, usually LCDs or projectors, is used to create a more high-resolution display than would otherwise be available. Because of the high density of these displays, the I/O rates necessary to display data are very high. Typically an array of PCs is used to provide inputs to the individual devices, or *tiles*, that make up the display. The tile reader benchmark is a tool for measuring the rate at which an I/O system can serve data to such a system.

The tile reader benchmark uses a 3 by 2 array of compute nodes (shown in Figure 9), each of which displays a fraction of the entire frame. By accessing only the tile data for its own display, the compute nodes exhibit a simple noncontiguous file access pattern. The six compute nodes each render a portion of the display with 1024 pixel by 768 pixel resolution and 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 between tiles. Each frame is 10.2 MBytes. This data is read into a contiguous buffer in memory. Our tests were conducted with a frame set consisting of 100 frames played back in sequence.

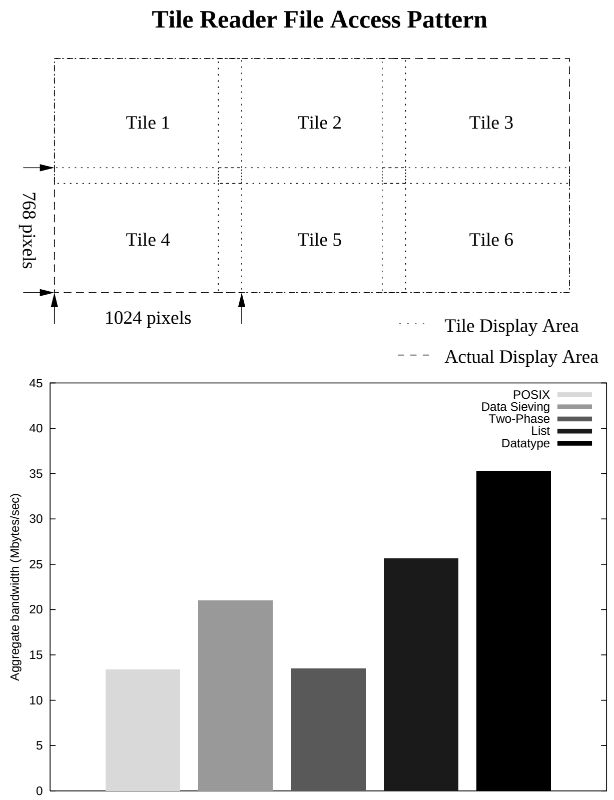

Fig. 9. Tile reader access pattern and performance

We can see in Figure 9 that datatype I/O is the clear winner in terms of performance for this benchmark, 37% faster than list I/O. The characteristics of the resulting reads using the optimizations tested are shown in Table II. The datatype I/O result is due to the combination of a single, concise I/O operation, no extra file data being transferred, and no data passing over the network more than once. In contrast, POSIX I/O requires 768 read operations, data sieving requires more than twice as much data to be read as is desired, two-phase I/O requires resending about 88% of the data read, and list I/O sends a list of 768 offset-length pairs as part of the requests (9 KBytes of total data in I/O requests from each client).

C. ROMIO Three-Dimensional Block Test

The ROMIO test suite comprises a number of correctness and performance tests. One of these, 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 10, has dimensions 600 x 600 x 600 with an element size of an integer (4 bytes on our test platform). Each process reads or writes a single one of these blocks. The memory datatype is contiguous.

Table III characterizes the resulting I/O patterns using our tested optimizations, and Figure 11 shows the results of our

	Desired Data per Client	Data Accessed per Client	# of I/O Ops per Client	Resent Data per Client	File Region Size Accessed
POSIX I/O	2.25 MB	2.25 MB	768	—	3 KB
Data Sieving I/O	2.25 MB	5.56 MB	2	—	4 MB
Two-Phase I/O	2.25 MB	1.70 MB	1	1.50 MB	1.70 MB
List I/O	2.25 MB	2.25 MB	12	—	3 KB
Datatype I/O	2.25 MB	2.25 MB	1	—	3 KB

TABLE II

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	File Region Size Accessed
8 Clients					
POSIX I/O	103 MB	103 MB	90,000	—	1200 B
Data Sieving I/O	103 MB	412 MB	103	—	4 MB
Two-Phase I/O	103 MB	103 MB	26	77.2 MB	4 MB
List I/O	103 MB	103 MB	1408	—	1200 B
Datatype I/O	103 MB	103 MB	1	—	1200 B
27 Clients					
POSIX I/O	30.5 MB	30.5 MB	40,000	—	800 B
Data Sieving I/O	30.5 MB	274.7 MB	69	—	4 MB
Two-Phase I/O	30.5 MB	30.5 MB	8	27.1 MB	4 MB
List I/O	30.5 MB	30.5 MB	626	—	800 B
Datatype I/O	30.5 MB	30.5 MB	1	—	800 B
64 Clients					
POSIX I/O	12.9 MB	12.9 MB	22,500	—	300 B
Data Sieving I/O	12.9 MB	206.0 MB	52	—	4 MB
Two-Phase I/O	12.9 MB	12.9 MB	4	12.1 MB	4 MB
List I/O	12.9 MB	12.9 MB	352	—	300 B
Datatype I/O	12.9 MB	12.9 MB	1	—	300 B

TABLE III

I/O CHARACTERISTICS OF THE ROMIO THREE-DIMENSIONAL BLOCK TEST

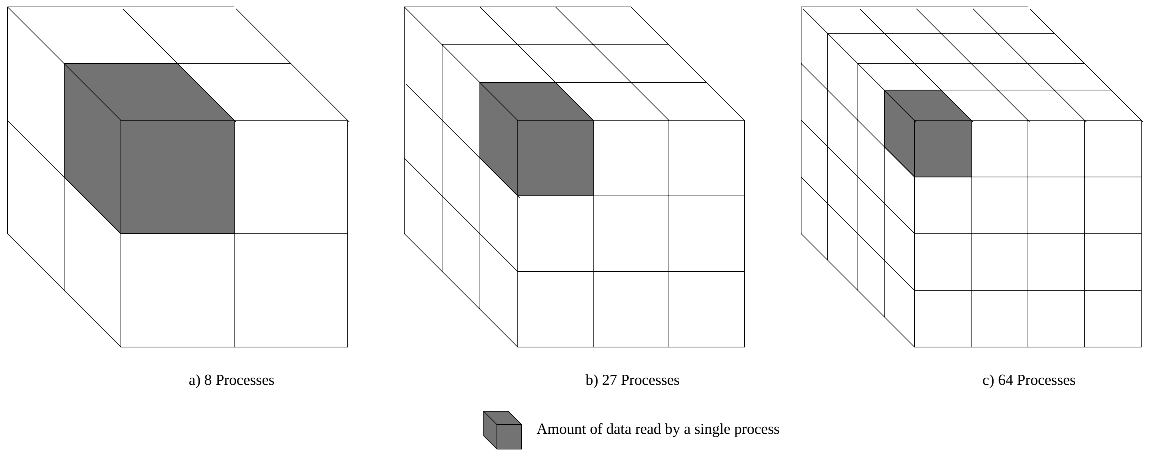

Fig. 10. Data distribution for three-dimensional block accesses

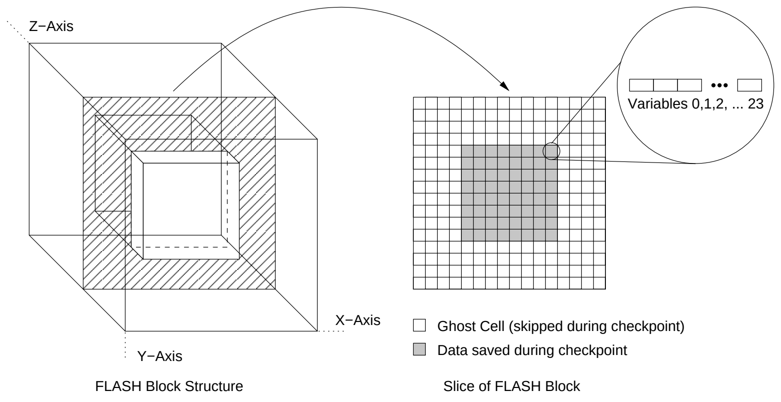

Fig. 12. FLASH memory layout

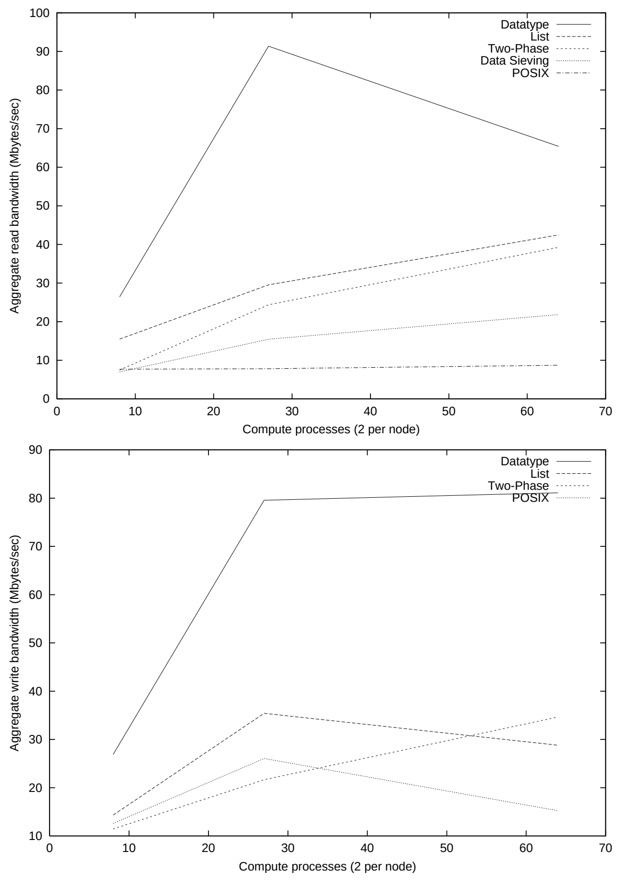

Fig. 11. Three-dimensional block read and write performance

tests. Again, datatype I/O is the clear performance winner; peak performance is more than double that of the next-best approach. Of note is the unusual drop in performance in the read case as number of processes increases. We believe that this is due to the increased overhead of offset-length list processing on the server side. Because the servers are the source of data, and clients are operating on a contiguous region of memory, any delays caused by list processing will directly impact performance. On the other hand, in the write case the servers are data sinks. Buffering in the TCP stack helps hide this inefficiency, although it might appear at larger numbers of processes. This overhead is not visible in the list I/O results because the number of I/O operations and the size of the I/O requests obscures this effect. A full-featured datatype I/O implementation that operated directly on the dataloop representation would likely not exhibit this behavior.

D. *FLASH I/O Simulation*

The FLASH code is an adaptive mesh refinement application that solves fully compressible, reactive hydrodynamic equations; the code was developed mainly for the study of nuclear flashes on neutron stars and white dwarfs [16]. Because the FLASH code has a very long execution time, checkpointing is a necessary component of the application. During checkpointing, blocks of data are reorganized from the in-memory organization, which includes ghost cells, into a new in-file organization. The in-file organization is stored by variable for convenience during post-processing. As a result the access pattern is noncontiguous in both memory and file.

Figure 12 shows the layout of data in the memory of FLASH processes. Each process holds 80 blocks. Each block is a three-dimensional array of data elements surrounded by guard cells. Each data element consists of 24 variables. When writing data we reorganize the data so that all values for variable 0 are stored first, then variable 1, and so on. 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 (at 2 clients) to 896 MBytes (at 128 clients).

Table IV provides the I/O characteristics of the test using the available optimizations. Figure 13 shows the results of these tests. This is the first test in which the memory datatype is noncontiguous; thus it is the first time that the overhead of list processing might affect the clients. We see this in both the list I/O and datatype I/O cases; both underperform at small numbers of clients. As the number of clients increases, the clients are eventually able to feed the servers adequately. At 96 processes, datatype I/O performance rises to nearly 40 Mbytes/sec, 37% faster than two-phase. This trend continues at higher numbers of processes. We would expect that

	Desired Data per Client	Data Accessed per Client	# of I/O Ops per Client	Resent Data per Client	File Region Size Accessed
POSIX I/O	7.50 MB	7.50 MB	983,040	—	8 B
Data Sieving I/O	—	—	—	—	—
Two-Phase I/O	7.50 MB	7.50 MB	2	7.5 MB * $\frac{n-1}{n}$	4 MB
List I/O	7.50 MB	7.50 MB	15,360	—	4 KB
Datatype I/O	7.50 MB	7.50 MB	1	—	4 KB

TABLE IV

I/O CHARACTERISTICS OF THE FLASH I/O SIMULATION (N IS THE # OF CLIENTS)

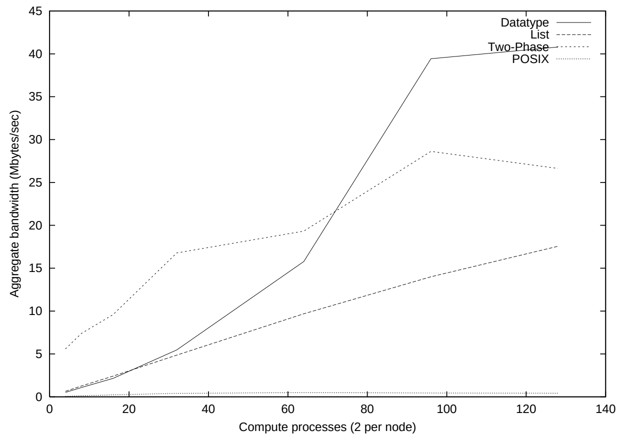

Fig. 13. FLASH I/O Performance

a datatype I/O system that operated directly on the dataloop representation would allow clients to more effectively push data to servers, resulting in improved performance at smaller numbers of clients. List I/O, because of the size and number of I/O requests, is not able to overtake two-phase for the tested numbers of processes. POSIX I/O is nearly unusable in this benchmark because it has to address 983,040 I/O operations of 8 bytes each to service this access pattern.

This case shows that two-phase I/O still has a place as an I/O optimization. Because data is not wasted and I/O accesses are large, two-phase I/O is able to provide good performance despite moving the majority of the data over the network twice.

VI. CONCLUSIONS AND FUTURE WORK

We have described our implementation of a new noncontiguous I/O method: datatype I/O. Our analytic comparison allowed us to explain the differences in performance that occur in various I/O access patterns with various I/O methods. The noncontiguous I/O benchmark suite verified our analytic comparison.

Datatype I/O provides the opportunity for extremely efficient processing of structured, independent I/O requests. Our tests show that this approach outperforms both list I/O and data sieving I/O in virtually all situations. Further, it supplants two-phase I/O as the preferred optimization in many cases as well. Datatype I/O in conjunction with the two-phase collective I/O optimization makes a strong MPI-IO optimization suite. We note that in almost every case POSIX I/O alone would result in a nearly unusable system from the performance perspective; these optimizations are a necessary part of scientific parallel I/O.

This prototype does not fully exploit the potential of the datatype I/O approach. We are implementing a more full-featured version of the approach in our second-generation parallel file system, PVFS2. This version will remove the creation of the I/O lists on both client and server, further widening the performance gap between datatype I/O and other optimizations. Datatype caching similar to that seen in some remote memory access implementations [17] could boost the performance of PVFS datatype I/O by further reducing I/O request overhead. Further optimization of the approach can be provided in ROMIO as well. Caching the dataloop representations of types locally would be one way to improve datatype I/O. Leveraging datatype I/O underneath two-phase I/O would boost performance of the collectives further.

ACKNOWLEDGMENT

This work was supported in part by National Aeronautics and Space Administration award number NGT5-158 and in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, SciDAC Program, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38.

REFERENCES

[1] “HDF5,” `http://hdf.ncsa.uiuc.edu/HDF5/`.

[2] J. Li, W. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, and R. Latham, “Parallel netCDF: A scientific high-performance I/O interface,” Mathematics and Computer Science Division, Argonne National Laboratory, Tech. Rep. ANL/MCS-P1048-0503, May 2003.

[3] R. Thakur, W. Gropp, and E. Lusk, “On implementing MPI-IO portably and with high performance,” in *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems*. ACM Press, May 1999, pp. 23–32.

[4] A. Ching, A. Choudhary, W. Liao, R. Ross, and W. Gropp, “Noncontiguous I/O through PVFS,” in *Proceedings of the 2002 IEEE International Conference on Cluster Computing*, September 2002.

[5] A. Ching, A. Choudhary, K. Coloma, W. Liao, R. Ross, and W. Gropp, “Noncontiguous I/O accesses through MPI-IO,” in *Proceedings of the Third IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2003)*, May 2003.

[6] J. Worringen, J. L. Traff, and H. Ritzdorf, “Improving generic noncontiguous file access for MPI-IO,” in *Proceedings of the 10th EuroPVM/MPI Conference*, September 2003.

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

[8] R. Thakur, A. Choudhary, R. Bordawekar, S. More, and S. Kuditipudi, "Passion: Optimized I/O for parallel applications," *IEEE Computer*, vol. 29, no. 6, pp. 70–78, June 1996. [Online]. Available: http://www.computer.org/computer/co1996/r6070abs.htm

[9] R. Thakur and A. Choudhary, "An extended two-phase method for accessing sections of out-of-core arrays," *Scientific Programming*, vol. 5, no. 4, pp. 301–317, Winter 1996. [Online]. Available: http://www.mcs.anl.gov/ thakur/papers/ext2ph.ps.gz

[10] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur, "PVFS: A parallel file system for Linux clusters," in *Proceedings of the 4th Annual Linux Showcase and Conference*. Atlanta, GA: USENIX Association, October 2000, pp. 317–327. [Online]. Available: http://www.mcs.anl.gov/ thakur/papers/pvfs.ps.gz

[11] W. B. Ligon and R. B. Ross, "Implementation and performance of a parallel file system for high performance distributed applications," in *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society Press, August 1996, pp. 471–480.

[12] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective I/O in ROMIO," in *Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*. IEEE Computer Society Press, February 1999, pp. 182–189. [Online]. Available: http://www.mcs.anl.gov/ thakur/papers/romio-coll.ps.gz

[13] R. Ross, N. Miller, and W. Gropp, "Implementing fast and reusable datatype processing," in *Proceedings of the 10th EuroPVM/MPI Conference*, September 2003.

[14] W. Gropp, E. Lusk, and D. Swider, "Improving the performance of MPI derived datatypes," in *Proceedings of the Third MPI Developer's and User's Conference*, A. Skjellum, P. V. Bangalore, and Y. S. Dandass, Eds. MPI Software Technology Press, 1999, pp. 25–30.

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

[16] 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*, vol. 131, p. 273, 2000.

[17] J. L. Traff, H. Ritzdorf, and R. Hempel, "The implementation of MPI-2 one-sided communication for the NEC SX-5," in *Proceedings of Supercomputing 2000*, November 2000.

Avery Ching is currently a Ph.D student under Professor Alok Choudhary at Northwestern University. He is studying high-performance and reliable storage techniques in parallel I/O technology for modern large-scale scientific applications.

Alok Choudhary is a Professor at Northwestern University. He has various interests in computer engineering including compilers, embedded and adaptive computing systems and power aware systems, high-performance databases, data warehousing OLAP and data mining, business intelligence and scientific applications, and parallel and high performance storage and I/O systems, including high performance computing, security, and computer architecture.

Wei-keng Liao is a research assistant professor in the Electrical and Computer Engineering Department at Northwestern University. He received a Ph.D. in computer and information science from Syracuse University in 1999. His research interests are in the area of high performance computing including parallel I/O, data mining, and data management for large-scale scientific applications.

Robert Ross received his Ph.D. in Computer Engineering from Clemson University in 2000. He then joined the Mathematics and Computer Science Division at Argonne National Laboratory. Rob's research interests are in system software for high-performance computing systems, in particular parallel file systems and I/O and message-passing libraries. Rob has been involved in Linux cluster computing and parallel file systems since 1995, when he first tested PVFS1 on early Beowulf systems at the NASA Goddard Space Flight Center.

William Gropp received his B.S. in mathematics from Case Western Reserve University in 1977, an MS in Physics from the University of Washington in 1978, and a Ph.D. in Computer Science from Stanford in 1982. He held the positions of assistant (1982-1988) and associate (1988-1990) professor in the Computer Science Department at Yale University. In 1990, he joined the numerical analysis group at Argonne, where he is a senior computer scientist and associate director of the Mathematics and Computer Science Division, a senior scientist in the Department of Computer Science at the University of Chicago, and a senior fellow in the Argonne-Chicago Computation Institute. His research interests are in parallel computing, software for scientific computing, and numerical methods for partial differential equations. He has played a major role in the development of the MPI message-passing standard. He is co-author of MPICH, the most widely used implementation of MPI, and was involved in the MPI Forum as a chapter author for both MPI-1 and MPI-2. He has written many books and papers on MPI including Using MPI and Using MPI-2. He is also one of the designers of the PETSc parallel numerical library and has developed efficient and scalable parallel algorithms for the solution of linear and nonlinear equations.