A New Flexible MPI Collective I/O Implementation

Kenin Coloma Avery Ching Alok Choudhary Wei-keng Liao
Electrical and Computer Engineering Department
Northwestern University
{kcoloma, aching, choudhar, wkliao}@ece.northwestern.edu

Rob Ross Rajeev Thakur
Mathematics and Computer Science Division
Argonne National Laboratory
{rross, thakur}@mcs.anl.gov

Lee Ward
Scalable Computing Systems
Sandia National Laboratories
lee@sandia.gov

Abstract

The MPI-IO standard creates a huge opportunity to break out of the traditional file system I/O methods. As a software layer between the user and the file system, an MPI-IO library can potentially optimize I/O on behalf of the user with little to no user intervention. This is all possible because of the rich data description and communication infrastructure MPI-2 offers. Powerful data descriptions and some of the other desirable features of MPI-2, however, make MPI-IO challenging to implement. By creating a new collective I/O implementation that allows developers to easily tinker and play with new optimizations or combinations of different techniques, research can proceed faster and be quickly and reliably deployed.

1 Introduction

MPI is undoubtedly a critical component of any cluster computing system without which the power of these systems would be far less accessible. As the number of processors in clusters and more specialized parallel systems continually grows, I/O is often a bottleneck for many applications ranging from climate modeling to computational physics. MPI-IO provides a portable means of accessing storage while also allowing for system specific optimizations. The MPI-2 collective I/O routines were included to preserve some semantic relationship between the I/O operations of several processes. Optimizations for this set of I/O functions can easily become very complex behind the scenes, sometimes limiting both the possible parameter variations and the number of developers and contributors. The ROMIO implementation of MPI-IO is very common, as it is distributed with Argonne National Laboratory's MPICH and several other MPI implementations. Working with ROMIO allows for a potentially broad impact on many high performance computing installations. Our new implementation seeks to provide similar behavior to ROMIO with respect to the two-phase I/O optimization while simultaneously delivering a cleaner code base and more avenues for research. The key goals for the new implementation are flexibility, developer friendliness, and performance. These compose an ideal research platform and important framework for further exploration. As an example, several variations on the two-phase I/O optimization are explored.

In Section 2 we describe collective I/O principles and the relevant MPI-IO specifications. Then in Section 3 we touch on related research, in particular the two-phase collective I/O optimization, before continuing on to describe the goals and design for the new implementation in Section 4. Implementation is discussed in Section 5, and performance for several benchmarks is analyzed in Section 6. Finally, in Section 7, the impact and benefits of the new implementation are considered.

2 Collective I/O

MPI-IO is a subset of the MPI-2 [7] specification. By using MPI derived datatypes, data can be transferred between complex file and memory layouts in single MPI-IO functions. A *memory datatype* describing how data should be accessed in memory is passed to the directly to the MPI-IO read and write functions. The *file view* for a particular process is described in a separate collective call with a *file datatype* and displacement. The file view sets out the accessible regions of a file by using the file datatype as a template and repeating it starting from the byte displacement. Figure 1 illustrates the construction of a file view and its components. The primary benefit of such a descriptive I/O inter-

1-4244-0328-6/06/$20.00

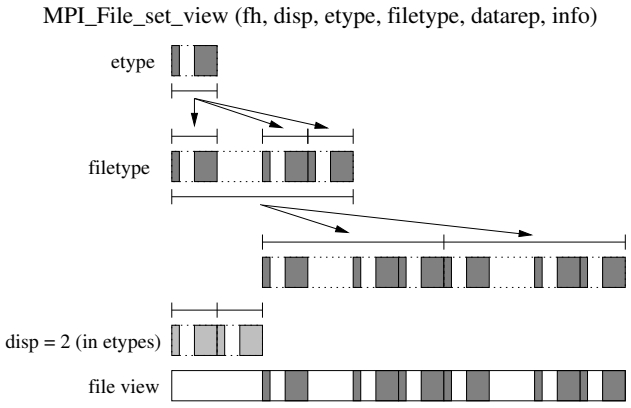

Figure 1. File views illustrated: Filetypes are built from *etypes* which themselves may be derived datatypes. The filetype access pattern is implicitly iterated forward starting from the *disp*. An actual count for the filetype is not required as it conceptually repeats forever, and the amount of I/O done is dependent on the buffer memory type and count.

face is the preservation of application intent. Rather than performing individual contiguous transfers between memory and file, the MPI library can capture a number of related I/O requests in an application and make optimizations to improve overall I/O performance. One such notable optimization is data sieving [14].

The collective I/O interfaces for MPI-IO retain the use of derived datatypes both for memory and file description, but they are also designed to allow cooperative I/O optimizations across processes by preserving higher-level access patterns. Collective I/O acknowledges that processes may be accessing storage in some meaningful way as a group. As the collective I/O functions are virtually identical to the *independent I/O* functions, they do not necessarily preclude the use of any independent I/O optimizations. As described in [15], the ROMIO [13] implementation uses both data sieving and the two-phase I/O method (a collective I/O optimization), developed by del Rosario [6]

The base collective I/O routines are:

```
MPI_File_set_view (fh, disp, etype,
    filetype, datarep, info)
MPI_File_read_all (fh, buf, count,
    datatype, status)
MPI_File_write_all (fh, buf, count,
    datatype, status)
```

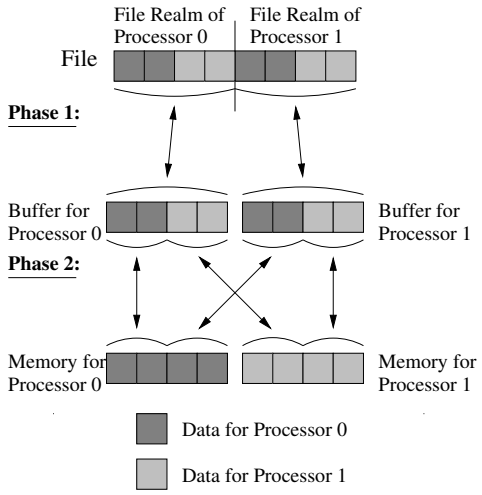

Figure 2. In the read case, data flows downward to processor memory. In the write case, data flows up to the file.

3 Related Work

Two-phase I/O conceptually consists of an I/O phase and a data exchange phase, but there is some additional incidental implementation specific communication. Figure 2 illustrates this idea. In the I/O phase, designated *I/O aggregators* in the collective communicator group are exclusively responsible for access to non-overlapping sections of the file, and perform I/O on behalf of all the processes. Determining these *file realm* assignments as well as which processes will be aggregators is left up to the implementation, which in turn may choose to defer to the user. For clarity's sake, *clients* will refer to a process in its capacity as the user source of the I/O requests, and aggregators refer to a process in its capacity as a point of I/O request gathering. All the processes involved in the collective I/O call are clients, but not all of those processes necessarily act as I/O aggregators. The communication phase moves file data to and from the appropriate processes. Whether the I/O call is a read or write determines the order of the two phases. In the read case, data is first read by the aggregators and then distributed to the requesting processes. In the write case, data is first sent to the aggregators and then written to the file by the aggregators. By combining I/O requests at the aggregators, overall I/O can be made more efficient. The number of I/O calls may be reduced, and the size of the aggregated remaining I/O calls may be increased. IBM's General Purpose File System (GPFS) has a feature called data shipping [11] which uses I/O aggregators in a similar way, but at the file system level, allowing for use in independent as well as collective I/O routines.

The disk directed collective I/O technique [9] allows I/O servers to optimize disk use across process requests. Because collective I/O is synchronized, I/O servers can accumulate the I/O requests from all the processes and then order block accesses in the best way possible. Performance is likely much better than if I/O requests had been handled in the order they happen to arrive in the collective I/O call.

The "Clusterfile" Parallel File System [8] uses both disk directed I/O and two-phase I/O together with a global file cache. This tight integration, as well careful attention paid to extracting parallelism from other subsystems such as memory and global file cache, result in significant performance improvements.

Persistent File Realms (PFRs)[5] use a modified two-phase I/O algorithm to maintain file system cache coherency within an MPI-IO application. Rather than optimizing file realm assignments for each individual collective read or write, file realms for the entire file are set during the first collective I/O routine, and remain unchanged between subsequent collective I/O calls until the file is closed. Since only one aggregator can access any given byte in the file (because file realms are non-overlapping), and all requests to that byte are funneled through that aggregator, all the processes' views of that byte are coherent. As a side benefit, though not insignificant, I/O locality is greatly improved since I/O aggregators are always accessing the same regions.

The datatype I/O method [3], illustrates the benefits of passing memory and file datatypes over the network describing non-contiguous regions rather than a one-to-one file mapping as in list I/O [2] where each contiguous file access is explicitly listed in some data structure. Operating on the datatypes directly instead of "flattening" the datatypes or entire accesses into offset/length pairs can also ease memory requirements of representing and storing data layout descriptions [16]. There is a threshold where storing datatypes does require more memory than storing the flat offset/length pairs representing the datatypes. The relative efficiency of each method, of course, depends on how exactly datatypes are stored, and the particular datatype in question.

4 Design and Motivation

In addition to the performance and portability goals of MPICH, the goals of the new collective I/O implementation are:

- Flexible Tuning
- Better Research Platform
- Easier Maintenance

The new two-phase I/O routines are implemented to reduce communication costs, scale, and take advantage of collective communication routines while retaining the current optimizations.

In addition to occasionally eliminating small inefficiencies, the new design allows for flexible tuning of more parameters. While this may not translate into something great for users, it provides systems developers and researchers with many more performance parameters. These parameters can then be tuned at a specific installation, or better yet, automatically determined at run-time with minimal user interaction.

Since supporting the rich descriptions of MPI-IO on any file system is a challenge, any implementation is bound to have a relatively high level of complexity. The necessity of adding in optimizations to realize the benefits of the MPI-IO interface also adds a degree of additional complexity. The new implementation exploits some of the opportunities for code reuse that were left out in the last implementation. The developers of the original ROMIO collective I/O functions had actually opted for less code reuse for a heavier emphasis on performance. While there is no way of completely eliminating complexity from the two-phase collective I/O code, the amount of code can certainly be reduced and modularized. The hope is that by making the code easier to study and understand, even more people will be able to use ROMIO as an I/O research platform, and successful ideas can be more easily and reliably integrated into the distribution.

5 Changes and Improvements

5.1 More Code Paths with Less Code

One potential advantage of the original collective I/O code is its tight integration with data sieving. Since data sieving is implemented directly in the collective I/O routines, the data sieve buffer also serves as the collective I/O buffer. From a performance perspective, this is very good, but from a maintenance and research perspective, it is challenging to have two separate data sieving implementations for collective I/O and independent I/O. The primary benefit of the integrated approach is one less buffer (and the resulting reduction of buffer copies) than the new code which uses the existing independent I/O internal calls and does not directly manage the data sieve buffer. In addition to the maintenance aspect, the new code can also easily leverage any of the optimizations to the internal independent I/O calls. For instance with a simple MPI hint, one could easily use the PVFS list I/O interface through MPI-IO [1], instead of data sieving beneath the MPI-IO collective I/O calls. Using list I/O would, incidentally, eliminate the double buffering issue since list I/O does not require an extra data buffer.

Furthermore, the entire collective I/O call needs not stick to only one optimization approach. Within one collective call, an aggregator may have to move data between the collective buffer and storage several times. Because the collective buffer is accessed in a single non-contiguous internal I/O call, it can potentially use a different optimization (preferably the best one) to do so each time. There is more fine-grained control over optimizations even within a single collective I/O call. Neither switching the I/O technique for accessing the collective buffer, nor the fine-grained control of those techniques can be easily added to the present code base. The last code path advantage of the new approach worth mentioning is the more efficient use of the collective I/O buffer. Unlike the data sieve buffer of the integrated approach, no unnecessary data (data sieving's gap data) resides in the collective buffer.

5.2 File Realms

File realm assignment is critical to the performance of two-phase I/O. These assignments affect the amount of data that needs to be redistributed across the network and the amount of data each aggregator needs to access. The current implementation found in ROMIO partitions the *aggregate access region* evenly among the I/O aggregators. The aggregate access region is the overall start and end file offset of the combined accesses of all the processes. This division and assignment is intended to keep the actual I/O responsibilities of the aggregators balanced so that in any given collective I/O, each aggregator ends up doing approximately the same amount of I/O. Note that this is a heuristic and not necessarily true for every access pattern. This method is best suited for combined accesses that are somewhat evenly distributed throughout the aggregate access region. Sparse clusters of data may create severe imbalances where some aggregators perform significantly more I/O than others. Since a collective I/O call can only be as fast as the slowest aggregator to return, this imbalance is a cause for concern.

By default, the new implementation uses the same aggregate access region based algorithm. The implementation, however, is built such that any arbitrary set of datatypes can describe the file realms. This feature necessitates a more general-purpose implementation since file realms are no longer assumed to be identical or even contiguous. In the general case, deciding what file realm a particular byte belongs to is no longer a simple $O(1)$ calculation, but a search. Since file realms in the new version are described using a datatype and a file offset (similar to a file view), one can easily plug in a new optimization function to determine the file realms in a completely different scheme. For instance, aggregator I/O loads could be better balanced. In hierarchical or very widely distributed systems, file realms could be distributed based on other system factors (such as network proximity). On a BG/L machine, it might be advantageous to ensure that aggregators sharing the same I/O node have adjacent file realms, thus improving cache locality on the I/O node.

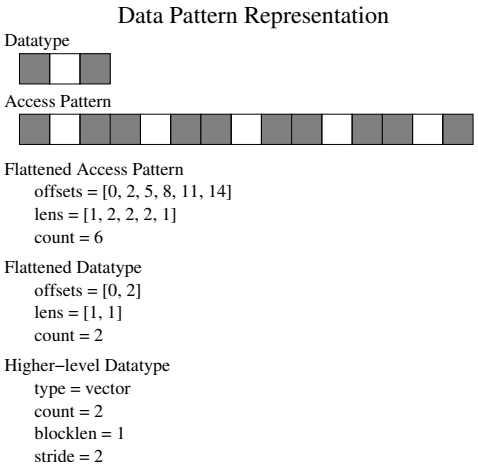

Figure 3. While the storage size of data patterns is important, one must consider any extra processing overhead involved. With respect to just storage size, the particular pattern determines which representation is best.

Because of the flexibility of file realm datatypes, implementing PFRs is a fairly easy task. The original collective I/O implementation had to be heavily modified in order to get persistent file realms to work. PFRs allow for many more states than the original code had been designed to handle. The primary difference with PFRs is file realms need to designate region assignments for the entire file, not just the region being accessed in one collective I/O call.

5.3 Storing, Processing, and Communicating Datatypes

Both data communication and I/O request communication are significantly overhauled. The basic terms used are:

- M is the total number of offset/length pairs for the access
- A is the total number of aggregators
- m_i is the number of offset/length pairs generated for aggregator i where $\sum_{i=0}^{A} m_i = M$
- D is the number of offset/length pairs to represent the datatype

The original I/O access communication is discussed first, and for simplicity, the discussion is in terms of a single client and multiple aggregators. The basic steps involved are as follows:

- client: flatten entire I/O access pattern into offset/length pairs (M)
- client: send appropriate offset/length requests to each aggregator (M)
- aggregator: receive applicable requests for its file realm from each client
- aggregator: processes each offset/length pair it receives (m_i)
- client: processes each of its offset/length pairs as aggregators make requests (M)

This basic algorithm is quite computationally efficient since its run-time is governed only by the number of offset/length pairs the client has from both the client perspective as well as the aggregator perspective. In the actual implementation, some user's offset/length pairs may be broken up, making the number of offset/length pairs dealt with more than M, but does not increase algorithmic complexity from $O(M)$. Space and communication time, however, are also linearly correlated with the number of offset/length pairs. For access patterns with numerous offset/length pairs, this memory cost can be prohibitive.

Potentially better alternative representations include storing the offset/length pairs of the datatypes themselves and storing the datatypes in an even higher level description as in Figure 3. Depending on the particular datatypes themselves, any one of the representations may be more space efficient than the other. The penalty for these more succinct access storage mechanisms, however, is more processing time. The new algorithm uses flattened datatypes and looks like this:

- client: flatten entire filetype into offset/length pairs (D)
- client: send flattened filetype to each aggregator
- aggregator: receive flattened filetype from each client
- aggregator: process flattened filetype looking for relevant data (M)
- client: process flattened filetype separately for each aggregator (MA)

The final step introduces extra work on the aggregator. Instead of simply being informed of the required accesses, it must calculate them itself. For the old scheme, between the client and aggregators, all offset length/pairs are processed in $O(M)$ times where M is the total number of offset/length pairs. In the new scheme, $O(MA)$ offset/length pairs are processed where A is the number of aggregators. The client needs to keep track of how each aggregator is progressing independently, so it must process its offset/length pairs once per aggregator. Conversely, each aggregator must basically process the client's entire access. While the aggregators can work in parallel, the client cannot, so the run time is $O(MA)$ as opposed to the $O(M)$ run time of the original collective I/O code. On the client side, a binary heap is used to mitigate this and achieve $O(MAlogA)$ time, and because of the iterative nature of datatypes, not all M offset/length pairs need to be processed. The latter technique, however, does not yield an algorithmic run-time change. Since the new file realms are simply datatypes, these are processed in a similar manner. The generalization of the interface results in a performance tradeoff.

While the original collective I/O code is very efficient from the computation standpoint of processing I/O requests, memory and communication requirements can become quite high for access patterns with many pieces. By only storing the offset/length pairs of the datatypes describing the access patterns and passing them around, memory and communication overheads are reduced. The cost of this design choice is the additional computation of aggregators processing each client's filetype.

5.4 Data Communication Optimizations

Although the original collective I/O code uses non-blocking communication frequently, it still performs all the communication at once. It first posts all the `MPI_Irecv`s, then posts all the `MPI_ISend`s, and then waits until all communication is complete. The new code uses either `MPI_Alltoallw` (an MPI-2 function) or non-blocking communication. `MPI_Alltoallw` allows aggregators to perform non-contiguous communication directly from the collective buffer. Similarly, it also allows consumer processes to do communication directly from the user buffers. Some recent high performance computing architectures, most notably IBM's BG/L, incorporate a separate network, or are at least highly optimized for, collective communication. In case such a specialized network is not available, the communication phase is also implemented with non-blocking communication in such a way as to overlap data communication with internal computation (e.g. file and memory address calculations).

6 Performance Testing

6.1 Machine Configuration

All of our tests were run on ASC Vplant at Sandia National Laboratories. Vplant is a large Linux cluster where each compute node has dual-processor 2.4 GHz Pentium 4 Xeons with 2 GB of main memory and Myrinet interconnect. Since, however, compiling MPICH2 with GM is tricky, TCP/IP was used. Tests were performed using the Linux 2.6.9 kernel running over a Lustres file system. Since the file system is shared between several clusters, each with many users, the best result out of five runs is reported for each experiment. While this methodology should better represent performance characteristics, it is also more susceptible to outliers and transient behaviors.

6.2 HPIO, Scalability, and Smarter Datatypes

HPIO [4], is a very flexible I/O benchmark useful for both performance and verification tests. HPIO builds regular datatypes that are characterized by a region size, count, and spacing. These datatypes are used to represent memory and/or file. The most common I/O patterns found in scientific computing involve non-contiguous access to files [10]. For Figure 4, data in both memory and file are non-contiguous. Regions of data are separated by 128 bytes of space, and there are 4096 regions per client. The amount of aggregate data accessed is between 2MB and 1 GB. The three I/O methods are the new collective I/O code run with a very succinct MPI "struct" datatype describing the non-contiguous patterns, the new collective I/O code with an MPI vector type explicitly enumerating the entire access, and finally, the original collective I/O code also with the MPI vector type. Since the original code flattens the entire access out, using the struct type with it makes no difference.

The consistently better performance of the new collective I/O with the struct datatype versus the new code with the vector datatype comes from several places. As shown in Section 5.3, using the shorter struct datatype should reduce the amount of data transferred over the network to exchange data layouts. Most of the difference in performance, however, is attributed rather to more efficient processing of the struct datatype than the full vector datatype. With the full vector datatype describing the entire access, clients must stop and evaluate each offset/length pair. An internal optimization allows processes to skip full datatypes in evaluating offset/length pairs, so that with such a succinct data description, processes may be able to skip many offset/length pairs while the runs with the vector datatype could not. As region sizes get larger, I/O time becomes a more significant factor, mitigating the benefits of the struct type. This is quite apparent when HPIO was run with only 8 aggregators.

Although the new collective I/O implementation fails to consistently match the performance of the older implementation, performance with the struct type is comparable in about half of the cases. MPE logging was used to identify the slow parts of the code, and it turns out that the main cause for the differences is the additional computational overhead tied directly to the number of aggregators. Double buffering between the separate collective and data sieve buffers also contributes to the performance differences. In the 8 aggregator case, the differences are pronounced because with fewer aggregators, each aggregator has to do more I/O, repeatedly going through both the collective and data sieve buffer for each I/O request to the file system. As the number of aggregators increases, the extra buffer copies become less significant, but the amount of computation increases.

With respect to performance, there is some room for improvement, and this is expected to come as the new code matures. The slight loss in performance is part of the tradeoff for opening more research opportunities by providing a relatively easy means for testing new optimizations as demonstrated in the following results sections.

6.3 Conditional Data Sieving with HPIO

In this experiment, the idea of conditional data sieving is introduced to the collective I/O routines using a simple metric. Aggregators can be directed to conditionally use different I/O techniques to access the file based on a characteristic or characteristics of the access pattern. In this test, two methods for writing contiguous data in the collective buffer to non-contiguous file space are tested. One uses a data sieving routine, and the other uses a naive routine that fulfills each contiguous request to the file with its own file system I/O call. In each graph, the datatype extent is held constant, and between graphs the file size is held constant at 1 GB. The amount of aggregate data accessed grows linearly along the X-axes from 32 MB to 1 GB. The initial theory tested is that a good means for determining when to use data sieving is based on the relative amount of useful data acquired in each file system I/O call. Our experiments in Figure 5 show that for HPIO, the extent of the datatype is more indicative of which method is more efficient. Naive I/O beneath two-phase is more suitable than data sieving for datatypes with larger extents. Conversely, data sieving is more efficient for smaller extents. The crossover is right around a 16 KB datatype extent. This threshold is very easily implemented so that user need not worry about where these crossover points are. The spikes most evident in the naive case are result of I/O access alignment. They occur in 4KB intervals, the page size for Lustre. The final spikes at the 100% points are a result of "the contiguous in memory to contiguous in file" code path being taken where compu-

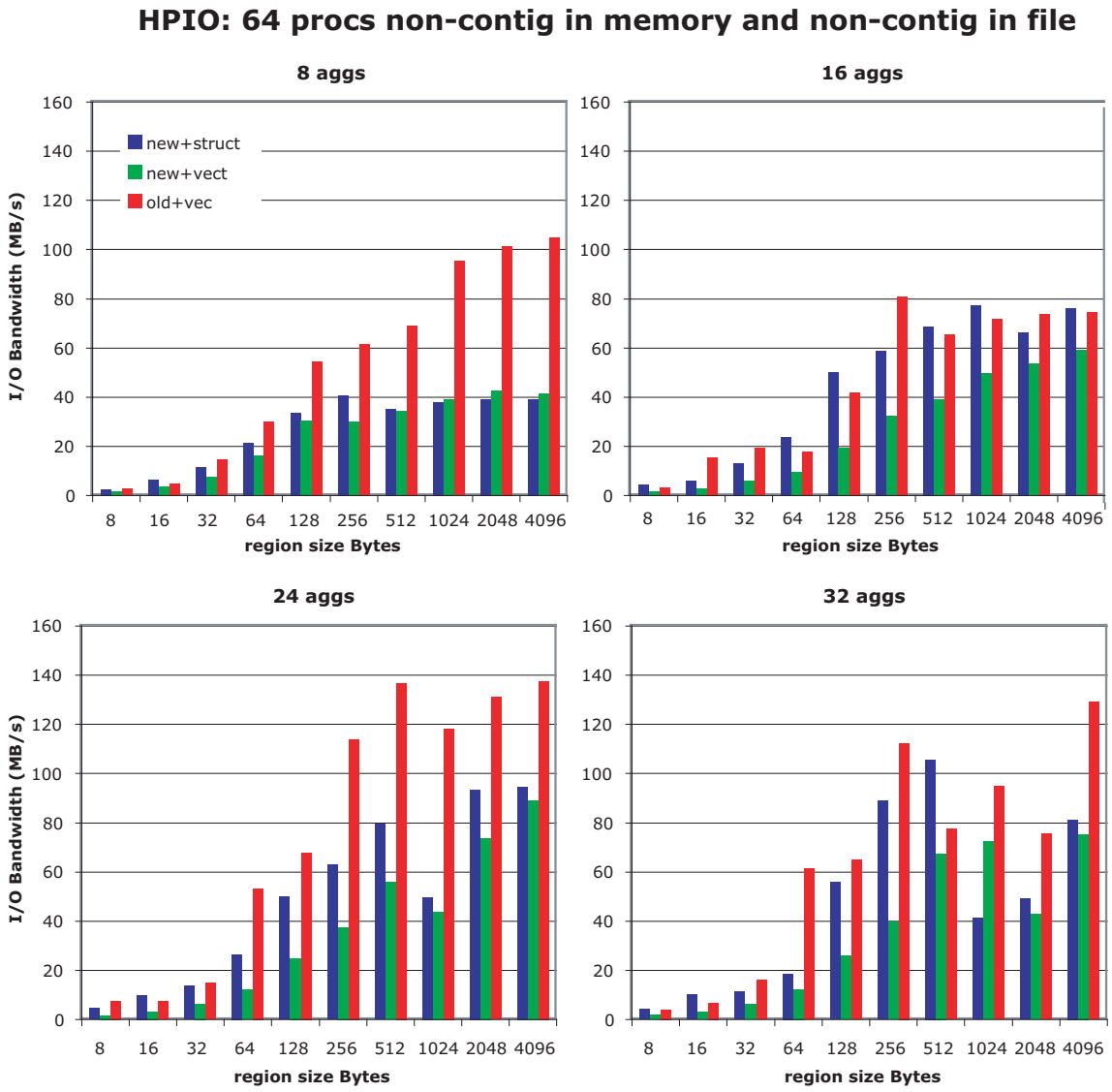

Figure 4. While the new collective I/O implementation provides comparable performance to the old implementation in many cases, in other cases, the new implementation fares significantly worse. This is due primarily the additional overhead required to process datatypes.

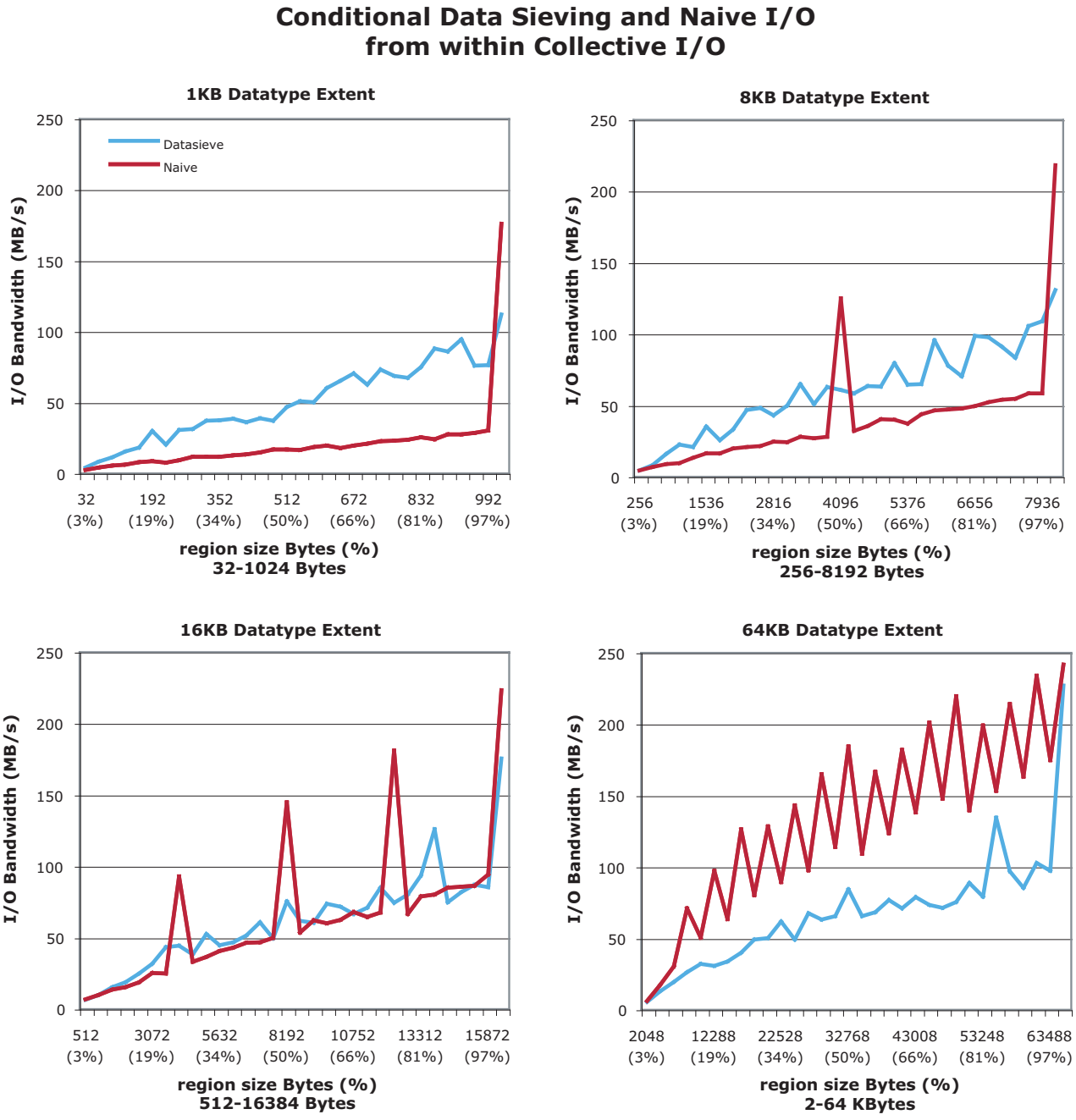

Figure 5. The percentages on the X-axes indicate the amount of useful data in the datatype relative to the entire extent of the datatype. This percentage is not as important a factor as the actual datatype extent in deciding whether or not to use data sieving or naive I/O to fill the collective buffer. The regularly spaced spikes are a result of I/O aligning nicely with the 4KB page size on the file system.

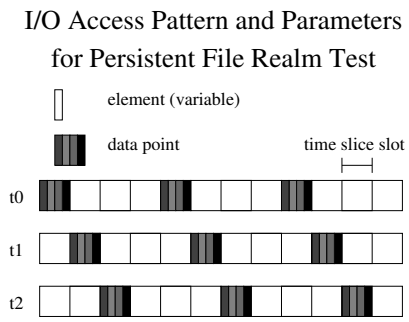

Figure 6. One non-contiguous collective write call is made per time step. Each data point is accessed with an interleaved pattern - in this case, four processes access an element each in every data point.

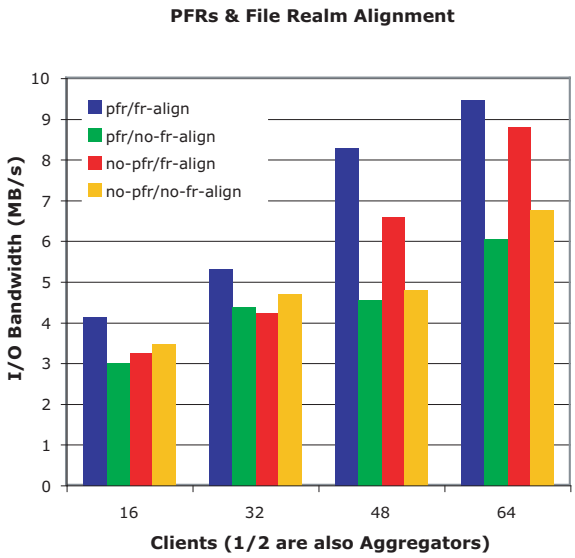

Figure 7. Careful alignment of file realms can ease the burden on file systems trying to maintain cache coherency and/or strict POSIX semantics.

tational overhead is significantly lower for two-phase I/O. The conditional formula used to determine the best I/O technique could easily accommodate the alignment behavior as well. The percentage of useful data in the data sieving operations may still become a factor for a very narrow band of extent values somewhere between 8KB and 16KB. These exact numbers, of course, are unique to the particular system used. This set of experiments merely demonstrates the potential for a number of parameter and combinational optimization studies enabled by the new implementation. More in-depth analysis of these behaviors is warranted, and it is much easier to do.

6.4 Persistent File Realms and File Realm Alignment

In high performance computing, files are often written but never read back in the same application. The PFR performance test code uses this principle to demonstrate the usefulness of an incoherent client-side file system cache in a write-only scenario. The analogy used in the construction of the program consists of a number of multi-variable data points distributed across the file where all the time steps for the same data point are kept together. The time steps can alternatively be thought of as a third spatial dimension, and seeing as the time steps are not indefinite, a fixed three-dimensional space with multiple variables makes sense as well. The size in the third dimension determines how many I/O calls are made. This access pattern is illustrated in Figure 6, and is similar to what may be generated by a higher-level library such as NetCDF [12]. Such a pattern ought to make good use of a client-side write-back cache.

File realm alignment is another easy optimization made in the new collective I/O routines. By aligning file realms with the file system stripe size (and more importantly, the page size), much of the overhead involved with locking and cache coherence in the underlying file system can be avoided. In this experiment on Lustre, file realms are aligned with the 2MB stripe size. The alignment is added as a new ROMIO hint, as there is really no portable way of determining the stripe size at run time.

One thing is clear from Figure 7, using PFRs with file realm alignment is a definite win. File realm alignment allows for nice accesses with respect to Lustre, and PFRs improve locality. Direct comparisons between each method are less straight forward, however. The nominal bandwidth numbers are low because the access pattern consists of sparse small data, and because data sieving is always on. The PFR code was run with 32 byte elements, 100 elements per data point, 2048 data points, and 32 time steps on Lustre. Under these options, only 6.5 MB of aggregate data is written in one collective write. Half of the total clients were also used as aggregators. Working on the assumptions that using file realm alignment is better than not, and that using PFRs is better than not, one would expect that using either of them should always be better than using neither. Experimentally, this is not always the case. Except for he results using just file realm alignment for 48 and 64 processes, using just one optimization is actually worse than not using any at all. This is an artifact of each of these cases gen-

erating slightly differing file realms. Without PFRs, the file realm sizes may remain the same as with PFRs, but the starting points move along the file with the actual access region, the PFR file realms are always anchored at byte zero of the file. PFRs establish a file realm assignment for all of the collective writes (because the file realms are anchored at 0), and without file realm alignment the Lustre lock manager is still engaged in lock granting and revocation. Similarly, without PFRs and with file realm alignment, the file realm alignment pays off for 48 and 64 processes, but with only 16 or 32 processes, the file realms that file realm alignment produces are just too imbalanced to effect a performance improvement.

7 Conclusions

Performance is certainly one area of weakness, but should get better with further development. By using datatypes to describe file realms, file realms can be further optimized for the general case as well as carefully tailored for specific applications, systems, and environments. Though the use of datatypes entails more processing overhead, for very large applications and large amounts of data, the network, memory, and I/O will likely be the predominant constraints. Better I/O aggregator load balancing is one obvious opportunity to leverage datatype based file realms, as well as more complex means of determining the appropriate I/O optimizations at run-time with little to no user intervention. The opportunity for improvements like these is the fundamental purpose of the new collective I/O implementation in its role as a powerful framework for future I/O research.

Acknowledgments

This work was supported in part by Sandia National Laboratories and DOE under Contract 28264, DOE contract B556691 // W-7405-ENG-48, DOE's SCiDAC program (Scientific Data Management Center), award number DE-FC02-01ER25485, DOE grant DE-FG02-05ER25683, NSF's NGS program under grant CNS-0406341, and NSF/CARP ST-HEC program under grant CCF-0444405.

References

[1] A. Ching, A. Choudhary, K. Coloma, W. K. Liao, R. Ross, and W. Gropp. Noncontiguous access through MPI-IO. In *Proceedings of the IEEE/ACM International Symposium on Cluster Computing and the Grid*, May 2003.

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

[3] A. Ching, A. Choudhary, W. K. Liao, R. Ross, and W. Gropp. Efficient structured data access in parallel file systems. In *Proceedings of the IEEE International Conference on Cluster Computing*, December 2003.

[4] A. Ching, A. Choudhary, W. K. Liao, L. Ward, and N. Pundit. Evaluating I/O characteristics and methods for storing structured scientific data. In *Proceedings of the International Parallel & Distributed Processing Symposium*, April 2006.

[5] K. Coloma, A. Choudhary, W. K. Liao, L. Ward, E. Russell, and N. Pundit. Scalable high-level caching for parallel I/O. In *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, April 2004.

[6] J. M. del Rosario, R. Bordawekar, and A. 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.

[7] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.

[8] F. Isaila, G. Malpohl, V. Olaru, G. Szeder, and W. Tichy. Integrating collective I/O and cooperative caching into the "clusterfile" parallel file system. In *Proceedings of the 18th Annual International Conference on Supercomputing*, pages 58–67, Sain-Malo, France, July 2004. ACM Press.

[9] D. 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.

[10] D. Kotz and N. Nieuwejaar. Dynamic file-access characteristics of a production parallel scientific workload. In *Proceedings of Supercomputing '94*, pages 640–649, Washington, DC, November 1994. IEEE Computer Society Press.

[11] J.-P. Prost, R. Treumann, R. Hedges, B. Jia, and A. Koniges. MPI-IO/GPFS, an Optimized Implementation of MPI-IO on top of GPFS. In *Proceedings of Supercomputing*, November 2001.

[12] R. Rew and G. Davis. The Unidata netCDF: Software for scientific data access. In *Proceedings of the 6th International Conference on Interactive Information and Processing Systems for Meterology, Oceanography and Hydrology*, February 1990.

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

[14] 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*, pages 182–189. IEEE Computer Society Press, February 1999.

[15] R. Thakur, W. Gropp, and E. Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 23–32, May 1999.

[16] J. Worringen, J. L. Traff, and H. Ritzdorf. Improving generic non-contiguous file access for MPI-IO. In *Proceedings of the 10th EuroPVM/MPI Conference*, September 2003.