IEEE International Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications
21-23 September 2009, Rende (Cosenza), Italy

Improving I/O Performance through the Dynamic Remapping of Object Sets

Jeremy Logan[1], Phillip Dickens[2]
[1] University of Maine, Orono, Maine, USA, jeremy.logan@maine.edu
[2] University of Maine, Orono, Maine, USA, dickens@umcs.maine.edu

Abstract - **Our research has been investigating a new approach to parallel I/O based on what we term objects. The premise of this research is that the primary obstacle to scalable I/O is the legacy view of a file as a linear sequence of bytes. The problem is that applications rarely access their data in a way that conforms to this data model, using instead what may be termed an object model, where each process accesses a (perhaps disjoint) collection of objects. We have developed an object-based caching system that provides an interface between MPI applications and a more powerful object file model, and have demonstrated significant performance gains based on this new approach. In this paper, we further explore the advantages that can be gained from using object-based I/O. In particular, we demonstrate that parallel I/O based on objects (termed parallel object I/O) can be dynamically remapped. That is, one application can output an object stream based on one object set, this can be captured and translated into a different object set that is more appropriate for another application. We demonstrate how such remapping can be accomplished, and provide an example application showing that using this technique can significantly improve I/O performance**.

Keywords - **Parallel I/O; High Performance Computing; Data-intensive applications; MPI-IO.**

I. Introduction

Large-scale computing clusters are increasingly being used to execute large-scale, data-intensive applications in several disciplines including, for example, high-resolution simulation of natural phenomenon, large-scale climate modeling, earthquake modeling, visualization/animation of scientific data, and distributed collaboration. The execution of such applications is supported by state-of-the-art file systems (e.g., Lustre [2], GPFS [18]) that provide tremendous aggregate storage capacity, and by parallel I/O interfaces that can interact with such file systems to optimize access to the underlying store. The most widely used parallel I/O interface is MPI-IO [4], which provides to the application a rich API that can be used to express complex I/O access patterns, and which provides to the underlying implementation many opportunities for important I/O optimizations. The problem, however, is that even with all of this hardware and software support, the I/O requirements of data-intensive applications are still straining the I/O capabilities of even the largest, most powerful file systems in use today. Thus new approaches are needed to support the execution of current and next-generation data-intensive applications.

There are many factors that make this problem, generally termed the *scalable I/O problem*, so challenging. The most often cited difficulties include the I/O access patterns exhibited by scientific applications (e.g., non-contiguous I/O [6, 7, 11]), poor file system support for parallel I/O optimizations [15, 16], strict file consistency semantics [12], and the latency of accessing I/O devices across a network. However, we believe that a more fundamental problem, whose solution would help alleviate all of these challenges, is the legacy view of a file as a linear sequence of bytes. The problem is that application processes rarely access data in a way that matches this file model, and a large component of the scalability problem is the cost of translating between the process data model and the file data model. In fact, the data model used by applications is more accurately defined as an *object model*, where each process maintains a collection of (perhaps) unrelated objects. We believe that aligning these different data models will significantly enhance the performance of parallel I/O for large-scale, data-intensive applications.

This research is attacking the scalable I/O problem by developing the infrastructure to merge the power and flexibility of the MPI-IO parallel I/O interface with a more powerful *object-based* file model. Toward this end, we have developed an *object-based caching system* that serves as an interface between MPI applications and object-based files. The object-based cache is based on MPI file views [3], or, more precisely, the intersections of such views. These intersections, which we term *objects*, identify all of the file regions within which conflicting accesses are possible and (by extension) those regions for which there can be no conflicts (termed *shared-objects* and *private-objects* respectively). This information can be used by the runtime system to significantly increase the parallelism of file accesses and decrease the cost of enforcing strict file consistency semantics and global cache coherence.

Previous research has shown that using the object-based caching system can lead to a significant increase in performance compared to native MPI-IO [13] for the FLASH-IO parallel I/O benchmark [1]. However, we did not at that time fully support the object file model. In particular, an object file created by one application could only be re-opened by another application with the same object set.

This issue can be thought of within the context of a producer/consumer problem. One application produces a set of objects (e.g., creates an object file) that another application requires (the consumer). However, the consumer requires a different object set. For example, a long running application may checkpoint its state information as a set of objects, terminate unexpectedly, and subsequently be restarted with a different number of processes. Another example would be when an application changes the file views upon which the current object set is based. More generally, an application's object set reflects the current file access patterns, and when access patterns change, new objects must be created that reflect such change. We refer to this as the *dynamic remapping* problem.

In this paper, we describe our approach to the dynamic remapping problem. It is based on the construction and utilization of interval trees, which store information about the current object set. Logically, what we refer to as a *translator* is placed between the producer and consumer applications, which utilizes the information stored in an interval tree to perform this translation.

During the course of this research it has become apparent that the ability to perform the dynamic remapping of object sets can provide the foundation for other important capabilities. For example, one emerging characteristic of next-generation scientific applications is their ability to adapt their behavior in response to changes in resource availability [10]. While there has been significant research investigating the remapping of the computational components of an application [14, 17], the issue of remapping its I/O component has been largely ignored. Dynamic remapping can also improve the performance characteristics of applications even when it would otherwise not be necessary to perform such remapping. This could occur, for example, when a producing application remaps the object set as it writes it to an object file such that it optimizes the performance of the consuming application.

The primary contribution of this paper is the development of a technique to perform dynamic remapping of parallel I/O and a demonstration of the performance enhancement that is available using this technique.

II. Background

MPI-IO is the IO component of the MPI standard [4] that was designed to provide MPI applications with portable, high performance parallel I/O. It provides a rich and flexible API that provides to an application the ability to express complex parallel I/O access patterns in a single I/O request, and provides to the underlying implementation important opportunities to optimize access to the underlying file system. It is generally agreed that the most widely used implementation of the MPI-IO standard is ROMIO [20-22, 24], which was developed at Argonne National Laboratory and is included in the MPICH2 [5] distribution of the MPI standard. ROMIO provides key optimizations for enhanced performance (e.g., two-phase I/O [9, 21]and data sieving[21-23]), and is implemented on a wide range of parallel architectures and file systems. The portability of ROMIO stems from an internal layer termed ADIO [22] (an Abstract Device Interface for parallel I/O) upon which ROMIO implements the MPI-IO interface. ADIO implements the file system dependent features, and is thus implemented separately for each file system.

A. MPI File Views

An important feature of MPI-IO is the file view [3], which maps the relationship between the regions of a file that a process will access and the way those regions are laid out on disk. A process cannot "see" or access any file regions that are not in its file view, and the file view thus essentially maps a contiguous window onto the (perhaps) non-contiguous file regions in which the process will operate. If its data is stored on disk as it is defined in the file view, only a single I/O operation is required to move the data to and from the disk. However, if the data is stored non-contiguously on disk, multiple I/O operations are required.

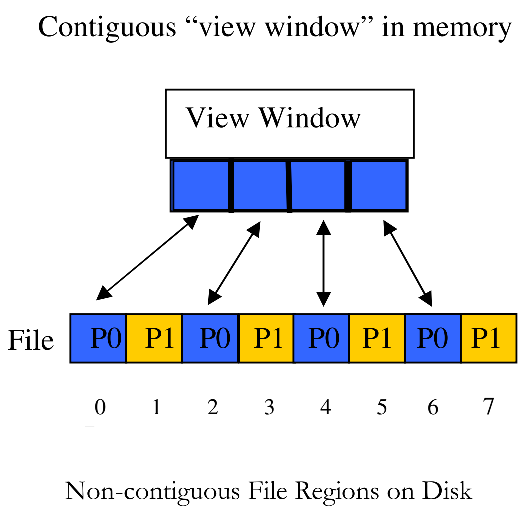
Contiguous "view window" in memory
View Window
File
P0
P1
P0
P1
P0
P1
P0
P1
0
1
2
3
4
5
6
7
Non-contiguous File Regions on Disk

Figure 1: The view window

Figure 1 depicts a file region in which two processes are operating, and the data for each is laid out non-contiguously on disk. The file view for Process P0 is shown, which creates a contiguous "view window" of the four data blocks it will access. Thus, the data model that P0 is using is a contiguous file region, which conflicts with the file data model. Because of these conflicting views, it will require four separate I/O operations to read/write its data from/to the disk. If it were stored on disk as it is used by P0, such data accesses would require a single I/O operation.

File views contain valuable information about file access patterns, and, when aggregated, show exactly those file regions in which contention is possible (there is overlap between file views), and, by extension, those regions for which contention is not possible.

B. Objects

Objects represent non-overlapping file regions that can be either private to a process (i.e., no other process will operate in that particular region), or shared by a set of processes. This distinction between shared objects and private objects has two very important ramifications: First, only shared objects must be locked, and such objects represent the minimum overlap of shared data. This completely eliminates false sharing and provides the maximum possible concurrency for data access. Second, such information can simplify the locking mechanism and significantly increase its performance. This is because each object manager knows exactly which processes can access its objects, and acts as a centralized lock manager for those processes. Thus contention for write locks, which can significantly reduce performance, is limited to the subset of processes that can access the objects being controlled by a given manager. In essence, this creates a set of centralized lock managers that are operating in parallel.

C. Object Based Caching System

The object-based caching system functions in the same manner as traditional file system caches except that it is objects rather than disk blocks that are cached. It takes an I/O request from an application process that is expressed in terms of a linear sequence of bytes and converts it to an equivalent request expressed in terms of objects. It then carries out the request either on its own (the object is private) or in collaboration with other object managers (the object is shared).

All of the processes that share a given file participate in the object cache for that file. The cache buffer consists of memory from the participating processes plus any available local disk space. The cache objects are created when a shared file is opened, and the objects and cache for that file are torn down when the file is closed. There is a local object manager for each process participating in the cache. Once the objects are created, they are distributed among the managers based on a cost model of assigning a given object to a given process. The local manager controls the meta-data for its objects and performs any object locking necessary to maintain global cache coherence. Once the objects are created, all subsequent I/O operations are carried out in the cache (except in the case of a sync() or close operation).

Metadata and locking responsibilities for a given object are maintained by (exactly) one of the object managers sharing the object. When a process wants to write into a shared object, the request is trapped by its local manager and forwarded to the appropriate lock manager. Once the lock has been acquired, the object can be written into the requesting processes' local cache, or the object can be modified and the updated object can be sent to the object manager that owns the object. In the first case, the write is performed and the writing process becomes the new manager for that object. In the latter case, ownership of the object is not changed. We are currently investigating the trade-offs associated with each approach.

III. Dynamic Remapping of Object Based I/O

Given an understanding of the basic system components we now focus on the dynamic remapping of object-based I/O. We first show how objects are created, followed by a discussion of how they are represented at runtime via interval trees. We then provide an example demonstrating how dynamic remapping can be performed.

D. Object Creation

Think of a file as represented by an integer line that extends from 0 to **n** – 1, where **n** is the number of bytes in the file. Given this representation of a file, a file view can be thought of as a set of intervals on this integer line, where each interval represents the endpoints of a file region in which the owning process will operate. These endpoints are obtained from the file views, and divide the integer line into a set of partitions termed *elementary intervals* [19]. Each file view can contain multiple intervals, and as more intervals are placed on the integer line, more elementary intervals are created. Once all of the intervals (of all file views) have been added to the line, each of the resulting elementary intervals corresponds to an object.

Figure 2 depicts object creation using this technique. It shows the file views of three processes, an integer line representing an 80-unit file, and how the endpoints associated with the file views cut the line. For example, the first interval associated with the file view of process P0 partitions the line into two segments, 0 -14 and 15 – 69. When the first interval associated with process P1 is added to the line, it creates three new partitions: 10 – 14, 15 – 25, and 26 – 29. The other partitions are similarly created, and an object is created for each resulting elementary interval.

As can be seen, each shared object encompasses exactly the overlapping file region within which contention can occur. It is also evident that objects are defined such that they cannot overlap.

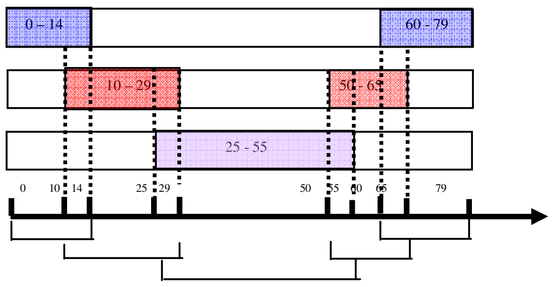

Figure 2: Object Creation. As intervals are added to the integer line new elementary elements are created.

E. Interval Search Trees

The advantage of creating objects in this manner is that it provides the basis for building an efficient search tree that can store and retrieve information about the object set. Because objects defined in this manner cannot overlap, it is possible to build a simple binary search tree, organized by object intervals, which can provide the needed functionality. Assuming that the tree is kept balanced, this approach would provide O(log n + r) lookup time, where n is the number of intervals (objects) stored in the tree, and r is the number of objects in the search result [8].

We needed slightly more functionality than that provided by a simple binary tree, and use instead an interval tree that stores the position on the integer line at each node, along with all objects containing the interval. Assuming the interval tree is kept balanced, it would have the same efficiency characteristics as those of a simple binary tree.

One of the most important issues with respect to binary trees is the method used to keep it balanced. A simple strategy for balancing is to choose a position for the root node that splits the integer line in half and recursively splits in half all of the sub-trees that are added. This would continue until all of the elementary intervals had been added. This simple strategy would result in a fairly balanced tree in cases where the file objects are evenly distributed, but has the potential for O(n) search performance in the worst case. There are many other options that can also be considered, ranging from probabilistic techniques such as random ordering of insertions, to the use of a self balancing tree such as a red-black tree or an AVL tree [8]. We are currently investigating the performance of these and other techniques.

F. Interval Tree Example

Figure 3 shows the interval tree that would be constructed to store the objects created in Figure 2. The tree nodes are created as described above, splitting the difference at each successive level of the tree. For example, the root of the tree divides the range of the file (0 – 79) in half, with the midpoint 39 being stored in the node. The root's left child divides the range between the beginning of the file and the position of the root node (0 – 39) in half, thus it has position 19.

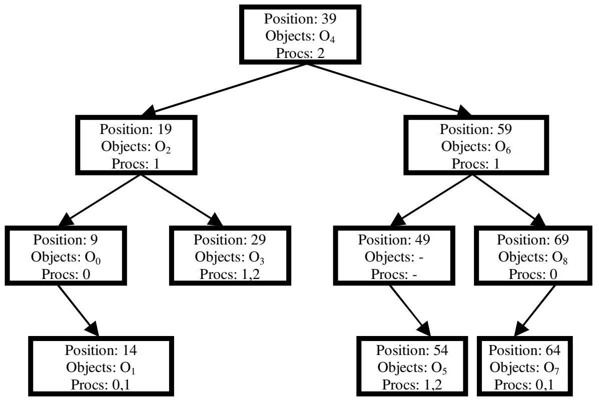

Figure 3: Interval Tree Example

Now consider how the interval tree would be searched to find all of the objects that contain the interval {18,35}. Starting at the root, the search interval is compared with the node's position. There are three possible outcomes from this comparison: the search range is entirely to the left of the node's position, the search range is entirely to the right of the node's position, or the search range contains the node's position. Since the root node's position is entirely to the right of the search interval, the right sub-tree may be safely pruned from the search as it cannot contain any objects in the search range. Any objects stored at the root node (in this case, only O_4) must be checked, as they may or may not intersect the search area. Since O_4 is found to overlap the search interval it is added to the result, and the left sub-tree is searched recursively.

The search of the root's left sub-tree starts at the node with position 19. Since position 19 falls within the search interval, the object at that node, O_2, must overlap the search interval (they have at least byte 19 in common), so it may be immediately added to the result. Since the node's position falls within the search interval, both left and right sub-trees must be searched recursively. The search moves to the node with position 9, which falls to the left of the search interval. O_0 is checked and ignored, as it does not fall within the search interval. The node with position 14 is searched recursively, and O_1 is also ignored. Finally, the node with position 29 is checked. Since 29 falls within the search interval, O_3 is immediately added to the result, completing the search.

G. Using Interval Trees for Dynamic Remapping

Logically, the dynamic remapping of object streams can be thought of as consisting of three components: a producer of objects, a consumer with a different object structure on the same file, and a translator that sits between the two. The translator is a distributed application that can be co-resident with either application (assuming multiple cores per node), or can reside on its own set of processors. There is no requirement that the translator has as the same number of processors as either application, although it simplifies the discussion to assume that there is a one-to-one mapping between an application process and a translator process. Each translator process is assigned the responsibility of a subset of the objects required by the consuming application.

In the general case, the translator would have an interval tree representing the object set of the producer and another representing the object set of the consumer. Assume a process of the producing application wrote an object to disk. This write would be trapped by the local cache manager and rerouted to the translator. For simplicity, assume the cache manager adds header information specifying the byte range of the object. When the translator receives the write request and the associated byte range, it searches the interval tree of the consumer application to determine the set of objects containing that interval. It would then determine the translator processes responsible for each of the returned objects, and send the file interval to each such process. A simple example may help clarify these ideas.

Consider the MPI-tile-reader benchmark that can be categorized as an example of the producer-consumer model. The producer application consists of a set of processes that generate a dense two-dimensional set of pixel data (tile-writers), and the consuming application consists of a set of processes that read the pixel data generated by the tile-writers and display the data on a tiled wallboard (tile-readers). The tiled wallboard consists of a set of individual monitors that together display the entire image. Adjacent monitors (in the horizontal and vertical directions) share a column of pixel data to help blend the individual components of the image into a smoother aggregate image.

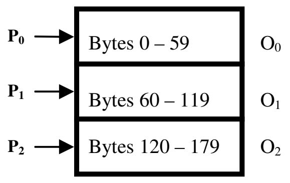

Figure 4: Each producer writes a single row.

For clarity of presentation, assume there are three tile-writers, four tile-readers, and a 60x3 two-dimensional grid of pixel data consisting of one byte per pixel. The object set for the three tile-writer processes is shown in Figure 4.

The tiled wall display consists of four monitors with a one-to-one mapping of tile-readers to monitors. Figure 5 depicts the object set required for the tile-readers. Tile-reader TR_0 is responsible for displaying bytes 0-19, 20-39, 60-79, and 80-99 on monitor M_0. Tile-reader TR_1 displays bytes 20-39, 40-59, 80-99, and 100-119 on monitor M_1. Thus bytes 20-39 and 80-99 are shared by TR_0 and TR_1 and displayed on their respective monitors. Similarly, bytes 60-79 and 80-99 are shared by TR_0 and TR_2, and displayed on the lower edge of monitor M_0 and the upper edge of monitor M_2. The object set associated with the tile-readers is also shown, and demonstrates how different applications can require different object sets on the same data.

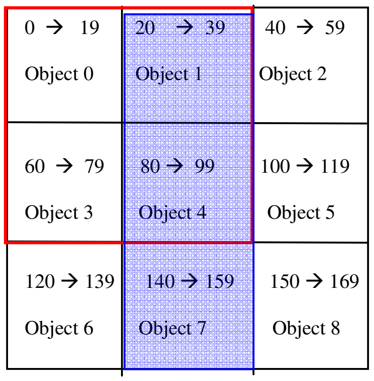

Figure 5. The tile-reader object set.

Figure shows the nine objects inserted into an interval tree that would be used in the transformation of the contiguous data written by the producer into the set of objects required by the consumer.

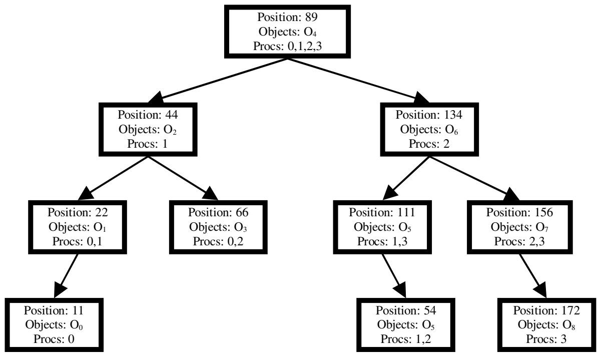

Figure 6: Tile reader objects positioned in an interval tree.

Figure gives an overview of the translator as it mediates between a producer application running on three processes and a consumer application running on four processes. The tile-writer's object set, consisting of three individual contiguous rows is shown on the left, and the tile-reader's object set is shown on the right. The translator processes receiving objects from the tile-writers determine where those objects fit into the tile-reader object set. The translator processes connected to the tile-readers build the new object set and write the new objects to the appropriate file for the tile-readers.

Each receiving translator process obtains an object from the corresponding process in the tile-writer application. That object's interval is used to perform a search of the interval tree representing the tile-reader's object set, which returns the set of objects that overlap that particular interval. For example, Object 0 of the tile-writer process extends from byte 0 to byte 59. When the interval 0-59 is searched, objects 0, 1, and 2 would be returned. The translator process would then send the object's data to the appropriate processes that are constructing the new object set.

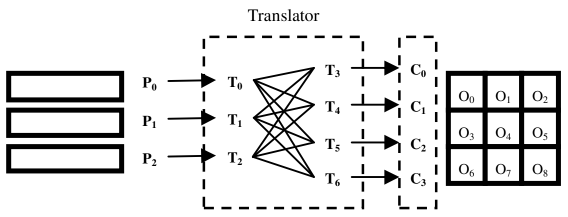

Figure 7: The translator architecture.

Finally, Figure shows the layout of objects in the tile-readers object file. Note that in this example the shared objects are replicated in the object file. This is because they are read-only objects and thus there is no concern about file consistency semantics. The advantage of this approach is that it allows a consumer process to read a single contiguous section of the file that contains all of the required data for that process, eliminating file contention and greatly increasing read performance. The disadvantage of this approach is that it requires more storage space. Thus the tradeoffs between more storage and better performance must be made on a case-by-case basis and are dependent upon the characteristics of the application.

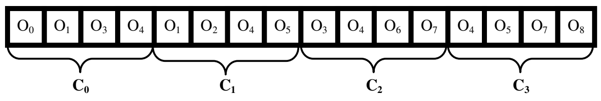
O_0 O_1 O_3 O_4 O_1 O_2 O_4 O_5 O_3 O_4 O_6 O_7 O_4 O_5 O_7 O_8
C_0
C_1
C_2
C_3

Figure 8: File layout after object transformation

IV. Experimental Design

We wanted to determine the magnitude of any performance gains that might be achieved by performing dynamic remapping of object streams. To begin to study this issue, we implemented the MPI-tile-reader benchmark with realistic data sets. In particular, each tile (monitor) had a display size of 1920 (width) by 1650 (height) and thus displayed a total of 3,168,000 pixels. Each pixel was represented by 32 bits with 8 bits each for red, blue, green and alpha (transparency). The tiles overlapped by 280 pixels in the x-direction and 150 pixels in the y-direction. We kept the number of monitors in the vertical and horizontal directions constant, and varied the dimension of the display between 2x2 and 10x10, thus varying the number of processors between 4 and 100.

For each configuration (2x2, 3x3, etc.), an object-based file was created to correspond with the data layout that would be required by each of the tile-reader processes. The objects belong to a particular tile-reader process were contiguous in the file. As noted, shared objects were replicated in the file.

The measurement of interest was the time required for the slowest process to read a single tile. We tested two strategies for reading. First, we left the tile-reader file as a linear sequence of bytes and used an unaltered version of MPI-IO (MPICH2 version 1.0.7) to read the file. Next, we used the same version of MPI-IO that was modified to support the object-based cache that read in the object-based file.

All experiments were conducted on Lonestar, a high-performance cluster consisting of 1300 Dell PowerEdge 1955 blades (nodes). Each node contains two Xeon Intel Duo-Core 64-bit processors running at 2.66 GHz and 8 GB of DDR-2 memory. The nodes are connected by an InfiniBand interconnect using a fat tree topology. Lonestar is attached to a 68 TB Lustre file system comprised of 16 Dell 1850 I/O data servers.

V. Results

The results are shown in Figure 9, which shows the time required to read a single tile using each strategy. As can be seen, the performance of the tile-readers utilizing the cache and object-based files was significantly better than that obtained using the native MPI-IO. In fact,

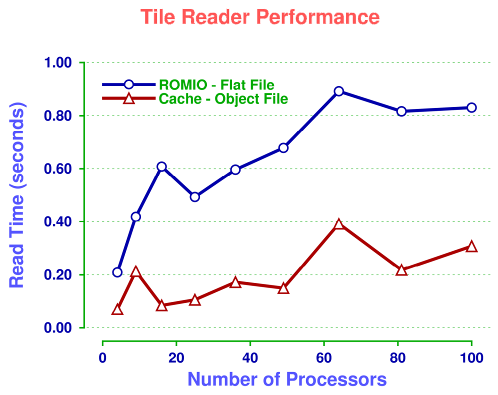

Figure 9. Comparison of MPI-Tile-Reader benchmark as a function of the approach used to create and read the pixel data. Note that ROMIO is the implementation of MPI-IO in the MPICH2 distribution used in this research.

performance was improved by between a factor of two and a factor of six for this particular benchmark. The performance of MPI-IO is limited in this case because the data being read is not contiguous in the file thus requiring multiple I/O requests to read in the file. Our technique, however, keeps the objects contiguous in the file resulting in a single I/O request.

VI. Discussion

These experiments were designed to get a handle on the magnitude of the performance gains made possible by remapping object data. These results represent the maximum gain in performance for this set of experiments because the translator was still under development at the time of this publication, and we thus created the object file manually to mirror how it would be created by the translator.

VII. Conclusions

In this paper, we have described our approach to performing object-based parallel I/O, and have demonstrated techniques by which dynamic remapping of object streams can be performed. We also showed that this technique is promising in its ability to improve I/O performance.

However, while these results are quite encouraging, the implementation costs of the transformations will be the ultimate determinant of the success of this approach. Given that a simple tree-based algorithm can be used to create such transformations, it appears that it could be implemented quite efficiently. We will provide data on the efficiency of the translator in future work.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. 0702748.

References

[1] FLASH I/O http://flash.uchicago.edu/~jbgallag/io_bench/

[2]. Cluster File Systems, Inc. http://www.clustrefs.com

[3]. MPI File Views http://www-unix.mcs.anl.gov/mpi/mpi-standard/mpi-report-2.0/node184.htm

[4]. MPI-2: Extensions to the Message-Passing Interface. Message Passing Interface Forum http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html

[5]. MPICH2 Home Page http://www.mcs.anl.gov/mpi/mpich

[6]. Avery Ching, Choudhary, A., Coloma, K., Liao, W.-k., et al., Noncontiguous I/O Accesses through MPI-IO. In the *Proceedings of the Third International Symposium on Cluster Computing and the Grid (CCGrid)*, (2002), 104-111.

[7]. Ching, A., Choudhary, A., Liao, W.-k., Ross, R., et al., Noncontiguous I/O through PVFS. In the *Proceedings of the 2002 IEEE International Conference on Cluster Computing (CLUSTER)*, (2002), 405-414.

[8]. Cormen, T., Leiserson, C., Rivest, R. and Stein, C. *Introduction to Algorithms, second edition.* The MIT Press, Cambridge, MA, 2001.

[9]. Dickens, P. and Thakur, R., A Performance Study of Two-Phase I/O. In the *Proceedings of the 4th International Euro-Par Conference*, (Southhampton, UK, 1998), 959-965.

[10]. Ghafoor, S., Haupt, T., Banicescu, I., Carino, R., et al., A Resource Management System for Adaptive Parallel Applications in Cluster Environments. In the *Proceedings of the The 6th International Conference on Linux Clusters: The HPC Revolution*, (Chapel Hill, North Carolina, 2005).

[11]. Isaila, F. and Tichy, W.F., View I/O: improving the performance of non-contiguous I/O. In the *Proceedings of the IEEE Cluster Computing Conference*, (Hong Kong).

[12]. Latham, R., Ross, R. and Thakur, R., The Impact of File Systems on MPI-IO Scalability. In the *Proceedings of the 11th European PVM/MPI Users' Group Meeting (Euro PVM/MPI 2004), Recent Advances in Parallel Virtual Machine and Message Passing Interface*, (2004), Lecture Notes in Computer Science, LNCS 3241, Springer, 87 - 96.

[13]. Logan, J. and Dickens, P., Using Object-Based Files for High-Performance Parallel I/O. In the *Proceedings of the IEEE International Workshop on Intelligent Data Acquisition and Advanced Computing Sysytems: Technology and Applications***.** (Dortmund, Germany, 2007).

[14]. Neema, S. and Ledeczi, A. Constraint Guided Self-Adaptation. *Self-Adaptive Software: Applications*, *LNCS 2614*. 39-51.

[15]. Ross, R., Latham, R., Gropp, W., Thakur, R., et al., Implementing MPI-IO Atomic Mode Without File System Support. In the *Proceedings of the 5th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2005)*.

[16]. Ross, R., Thakur, R. and Choudhary, A. Achievements and Challenges for I/O in Computational Science. *Journal of Physics: Conference Series (SciDAC 2005)*, *16*. 501 - 509. 2005.

[17]. Schmidt, D.C., Box, D.F. and Suda, T. ADAPTIVE: A Dynamically Assembled Protocol Transformation, Integration, and eValuation Environment. *Journal of Concurrency: Practice and Experience*, *5* (4). 269-286. June 1993.

[18]. Schmuck, F. and Haskin, R., GPFS: A shared-disk file system for large computing clusters. . In the *Proceedings of the Conference on File and Storage Technologies*, (IBM Almaden Research Center, San Jose, California).

[19]. Stewart, J. CSC378: Interval Trees www.dpg.toronto.edu/people/JamesStewart/378notes/22intervals

[20]. Thakur, R., Gropp, W. and Lusk, E., An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. In the *Proceedings of the Proc. of the 6th Symposium on the Frontiers of Massively Parallel Computation*.

[21]. Thakur, R., Gropp, W. and Lusk, E., Data Sieving and Collective I/O in ROMIO. In the *Proceedings of the Proc. of the 7th Symposium on the Frontiers of Massively Parallel Computation*, 182-189.

[22]. Thakur, R., Gropp, W. and Lusk, E., On Implementing MPI-IO Portably and with High Performance. In the *Proceedings of the Proc. of the Sixth Workshop on I/O in Parallel and Distributed Systems*, 23-32.

[23]. Thakur, R., Gropp, W. and Lusk, E. Optimizing Noncontiguous Accesses in MPI-IO. *Parallel Computing*, *28* (1). 83-105. January, 2002.

[24]. Thakur, R., Ross, R. and Gropp, W. Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation, Technical Memorandum ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, Revised May 2004.