Using MPI File Caching to Improve Parallel Write Performance for Large-Scale Scientific Applications

Wei-keng Liao, Avery Ching, Kenin Coloma, Arifa Nisar, and Alok Choudhary
Electrical Engineering and Computer Science Department
Northwestern University
Evanston, Illinois 60208-3118
{wkliao,aching,kcoloma,ani662,choudhar}@ece.northwestern.edu

Jacqueline Chen
Combustion Research Facility
Sandia National Laboratories
Livermore, California 94551-0969
jhchen@sandia.gov

Ramanan Sankaran and Scott Klasky
National Center for Computational Sciences
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831-6008
{sankaranr,klasky}@ornl.gov

ABSTRACT

Typical large-scale scientific applications periodically write checkpoint files to save the computational state throughout execution. Existing parallel file systems improve such write-only I/O patterns through the use of client-side file caching and write-behind strategies. In distributed environments where files are rarely accessed by more than one client concurrently, file caching has achieved significant success; however, in parallel applications where multiple clients manipulate a shared file, cache coherence control can serialize I/O. We have designed a thread based caching layer for the MPI I/O library, which adds a portable caching system closer to user applications so more information about the application's I/O patterns is available for better coherence control. We demonstrate the impact of our caching solution on parallel write performance with a comprehensive evaluation that includes a set of widely used I/O benchmarks and production application I/O kernels.

1. INTRODUCTION

Modern scientific applications often run long and periodically write snapshots of the current computational state to checkpoint files. Checkpoint files are later used for post-simulation data analysis such as visualization of the data evolution over finite time steps. They are also used for application restart when the program is unexpectedly terminated or gone awry. Once written, checkpoint data is never accessed again for the rest of the application run. Such write-only checkpointing patterns have come to dominate the overall I/O activities in many large-scale applications; therefore, designing efficient techniques for addressing this is very important.

The write-behind strategy is a well-known technique used by operating system designers as a way to speed up sequential writes [22]. The basic algorithm accumulates multiple small writes into large contiguous file requests in order to better utilize the network bandwidth. The implementation of write-behind is often part of a file system's client-side caching component. File caching has achieved significant success in distributed environments where files are rarely accessed by more than one client concurrently. However, in parallel environments where applications employ multiple processes to solve a single problem, parallel I/O on shared files becomes more frequent. In a parallel application, the problem domain is often represented by a set of global data structures, such as multi-dimensional arrays. These global data structures are partitioned among the processes and each process operates on the data in its sub-domain. When writing the global data structures in files, it is desirable to maintain their global canonical order. Combining the partitioned data structures from multiple clients involves concurrent write operations to a shared file. For shared-file I/O, file caching introduces the cache coherence problem. It occurs when changes to a client's local copy of cached data do not propagate to other copies in a timely manner, leaving caches in an incoherent state. Coherence control commonly involves bookkeeping of cache status at file servers and invoking client callbacks as necessary to flush dirty data. Such mechanisms require a lock as part of each read/write request to ensure atomic access to cached data. While forcing a lock request for every I/O call guarantees the desired outcome, it can easily limit the degree of I/O parallelism for concurrent file operations. We believe that by moving the cache system closer to user applications, information about applications' I/O patterns can be used for better coherence control. Furthermore, the write-behind optimization can be more effective when the write-only mode is known in advance.

The Message Passing Interface (MPI) standard defines an application programming interface for developing parallel programs that explicitly use message passing for inter-process communication [15]. MPI version 2 extends the interface to include, among other things, file I/O operations [16]. MPI I/O focuses on the functions for concurrently access-

SC07 November 10-16, 2007, Reno, Nevada, USA
Copyright 2007 ACM 978-1-59593-764-3/07/0011...$5.00

ing shared files. In practice, very few of today's large-scale scientific applications actually use MPI I/O for data access. Instead, the majority of MPI applications simply use the POSIX I/O functions (e.g. open, read, write, close) and let each process access a unique file independently for performance reasons. Poor shared file performance is typically caused by the inefficient file system lock management. In many parallel file systems, lock granularity is at the file block, instead of the byte level. Block granularity simplifies lock management but causes false sharing when two concurrent I/O requests access the same file block, regardless of whether there are any overlapping bytes. For block based systems, good parallel I/O performance can only be obtained when requests are carefully aligned with lock boundaries: a rarity in real applications

We prototyped a user-level file caching layer as part of an MPI I/O library by incorporating the MPI communicator concept to enable process collaboration for caching [12, 11]. In the rest of the paper, we refer to this work as MPI-IO caching. One immediate benefit of this design is that the file system can pass consistency control responsibilities to the caching system. We believe that knowing the group of clients that will later access a shared file is the first step in tracking incoherent cache state effectively. We use an I/O thread in each MPI process to handle local file caching and remote cache page access. All I/O threads communicate with each other for cache coherence control. In our design, a file is logically divided into equally sized blocks that are the minimal caching unit. The cache metadata for the blocks is cyclically distributed among all processes to avoid centralized management. A lock protocol is developed to ensure the integrity of the cache data and metadata. Since data are buffered at this caching layer between the application and file system, there is an opportunity to reorganize the I/O requests to align with the system lock boundaries and minimize lock contention. In our experiments, this alignment provides a significant performance improvement.

To specifically address the write-only I/O pattern, we designed a two-stage write-behind buffering scheme. This scheme is built based on the implementation of MPI-IO caching and retains the general benefits of caching, but without the coherence control overhead. This scheme has two requirements: the file must be opened in write-only mode and MPI atomic I/O mode be disabled. These requirements are met with data checkpoints in modern scientific applications since they only consist of non-overlapping write operations. The first stage write-behind accumulates write data into local buffers and flushes the buffers to the second-stage global buffers if they are full. The location of global buffers in the second stage is based on a cyclic file block assignment among the MPI processes, so data for a particular block is always flushed to the same process. Once the global buffers are full, they are written to the file system. Just as in MPI-IO caching, this scheme also enables write alignment with the file system lock boundaries.

In general, client-side file caching enhances I/O performance under two scenarios: I/O patterns with repeated accesses to the same file regions, and patterns with large numbers of small requests. For the former, caching reduces the number of data transfers between clients and servers. In this paper, we do not consider this pattern since it is not commonly seen in today's parallel applications. We instead focus on the latter I/O pattern and use the BTIO, FLASH I/O, and S3D I/O benchmarks to present a comprehensive performance evaluation. We examine both the MPI-IO caching and write-behind approaches on two parallel machines running the Lustre and IBM GPFS file systems. The performance results demonstrate that the proposed methods succeed in using a data buffering layer to align write requests with the file system lock boundaries and effectively reduce file system lock contention that otherwise appears in unaligned accesses.

The rest of the paper is organized as follows. Section 2 discusses background information and related work. MPI-IO caching is described in Section 3. The design and implementation of the two-stage write-behind method are presented in Section 4. Performance results are given in Section 5 and the paper is concluded in Section 6.

2. BACKGROUND AND RELATED WORK

Many production MPI applications do not use MPI I/O functions for file access. Instead, I/O programming often consists of POSIX I/O interfaces in a style of accessing one file per process. This is done primarily because of the poor shared-file I/O performance of today's file systems. This one-file-per-process approach, however, produces an extremely large number of files, which creates a daunting technical challenge for file system design and post-simulation data analysis. For instance, when a production run of an application uses 1000 processes and takes 100 checkpoints, the one-file-per-process policy creates 100,000 files in total compared to 100 files if data are written into a shared file at each checkpoint. Since today's parallel file systems use a comparatively small number of metadata servers, as the number of processes and files increases dramatically, it is very plausible that the heavy metadata workload may disrupt the overall file system performance. Additionally, during post-simulation data analysis, these sub-array files often need to be merged into global arrays in a canonical format. This step is often costly during the post processing. Furthermore, the raw-process checkpoint files created by a given number of MPI processes may not be directly useful for program restart with a different number of processes.

2.1 MPI I/O

MPI I/O inherits two important MPI features: MPI communicators define a set of processes for group operations and MPI derived data types describe complex memory layouts. A communicator specifies the processes that can participate in a collective MPI operation for both inter-process communication and I/O requests to a shared file. For file operations, file open requires an MPI communicator to indicate the group of processes accessing the file. In general, MPI I/O data access operations can be split into two types: collective I/O and independent (non-collective) I/O. Collective operations require all processes in the communicator to participate. Because of this explicit synchronization, many collective I/O implementations may exchange access information among all processes to generate a better overall I/O strategy. An example of this is the two-phase I/O technique [4]. Two-phase I/O is used in ROMIO, a popular MPI I/O implementation developed at Argonne National Laboratory

[25]. Independent I/O, in contrast, requires no synchronization and makes any collaborative optimization very difficult.

Two-phase I/O consists of an I/O phase and a communication phase. First, an aggregate access region is calculated as a contiguous range that covers all the I/O requests from all MPI processes. *File domains*, contiguous regions for which a processes are exclusively assigned, are then defined and calculated by evenly dividing the aggregate access region by the number of designated I/O aggregators. The division is done at the byte range granularity. I/O aggregators can be defined as all of or some subset of the MPI processes that opened the shared file collectively. In the I/O phase, each MPI aggregator makes read/write calls on behalf of all processes to the file system for the requests within its file domain. In the communication phase, data is distributed either to processes from aggregators (read case) or vice versa (write case). The file domain calculation currently used in ROMIO balances I/O workload among the aggregators for I/O evenly distributed across the aggregate access region, but may not always result in the best I/O performance.

MPI file view can describe the data partitioning of a global data structure among the processes and map a process' sub-array to the global array in the file. A process' partition is described with an MPI derived datatype created with the starting file offsets of the sub-array relative to the global array, the sub-array dimensionality, and the stride size for repeating non-contiguous segments. The construction of a derived datatype can also be nested. File views are of particular importance for collective I/O operations because they are exchanged between processes to help improve I/O performance. File views can also be used together with MPI independent I/O. Although the most popular programming style of using independent I/O is similar to the one using POSIX I/O, its performance has compared poorly to collective I/O in several studies [4, 23, 26].

2.2 File Locking in Parallel File Systems

Most file systems adhere to the POSIX standard: two POSIX requirements on I/O consistency and atomicity often significantly degrade shared-file I/O performance. I/O consistency is an issue that arises when client-side file caching is enforced by file system's cache coherence control. POSIX atomicity requires that individual write calls are either entirely visible or completely invisible to any read call [9]. A common solution to meet these two requirements uses file locking to guarantee exclusive access to file regions. Once lock contention occurs, the degree of I/O parallelism is severely limited for concurrent file operations.

Client-side file caching is supported in many parallel file systems; for instance, IBM's GPFS [17, 21] and Lustre [13]. By default, GPFS employs a distributed token-based locking mechanism to maintain coherent caches on nodes. Lock tokens must be granted before any I/O operation is performed [18]. Distributed locking avoids the obvious bottleneck of a centralized lock manager by making a token holder a local lock authority for granting further lock requests to its corresponding byte range. A token allows a node to cache data that cannot be modified elsewhere without first revoking the token. Lock granularity for GPFS is the disk sector size. IBM's MPI I/O implementation over AIX operating system uses the data shipping mechanism: files are divided into equally sized blocks, each of which is bound to a single I/O agent, a thread in an MPI process. The file block assignment is done in a round-robin striping scheme. A given file block is only cached by the I/O agent responsible for all accesses to this block. All I/O operations must go through the I/O agents which then "ship" the requested data to the appropriate processes. Data shipping maintains cache coherence by allowing at most one cached copy of file data among agents. The Lustre file system uses a slightly different distributed locking protocol where each I/O server manages locks for the stripes of file data it stores. The lock granularity for Lustre is the file system page size. If a client requests a lock held by another client, a message is sent to the lock holder asking it to release the lock. Before a lock can be released, dirty cache data must be flushed to the servers. Both Lustre and GPFS are POSIX compliant file systems and therefore respect POSIX I/O atomicity semantics. To guarantee atomicity, file locking is used in each read/write call to guarantee exclusive access to the requested file region. On parallel file systems like Lustre and GPFS where files are striped across multiple I/O servers, locks can span multiple stripes for large read/write requests. Lock contention due to atomicity enforcement can significantly degrade parallel I/O performance [19].

2.3 Cooperative Caching and Active Buffering

Cooperative caching has been proposed as a system-wise solution for caching and coherence control [3]. Multiple clients coordinate to relay requests not satisfied by one client's local cache to another client. Systems that use cooperative caching include PGMS [27], PPFS [8], and PACA [2]. The Clusterfile parallel file system integrates cooperative caching and disk direct I/O for improving MPI collective I/O performance [10]. Cooperative caching generally requires changes in the file system at both client and server.

Active buffering is an optimization for MPI collective write operations [14]. It accumulates write data into a local buffer and uses an I/O thread to perform write requests in the background. I/O threads dynamically adjust the size of local buffers based on the available memory space. For each write request, the main thread allocates a buffer, copies the data over, and appends this buffer to a queue. The background I/O thread later retrieves the buffers from the head of the queue, writes the buffered data to the file system, and then releases the buffer space. Although active buffering demonstrates a significant performance improvement, it is limited to MPI collective I/O and does not address the potential for lock contention in the underlying file systems.

3. CLIENT-SIDE FILE CACHING FOR MPI-IO

As it is designed for MPI applications, our caching system uses the MPI communicator supplied to the file open call to identify the scope of processes that will collaborate with each other to perform file caching. To preserve the existing optimizations in ROMIO, like two-phase I/O and data sieving, we incorporate our design in the Abstract Device I/O (ADIO) layer where ROMIO interfaces with underlying file systems [24]. To enable the collaboration of MPI processes,

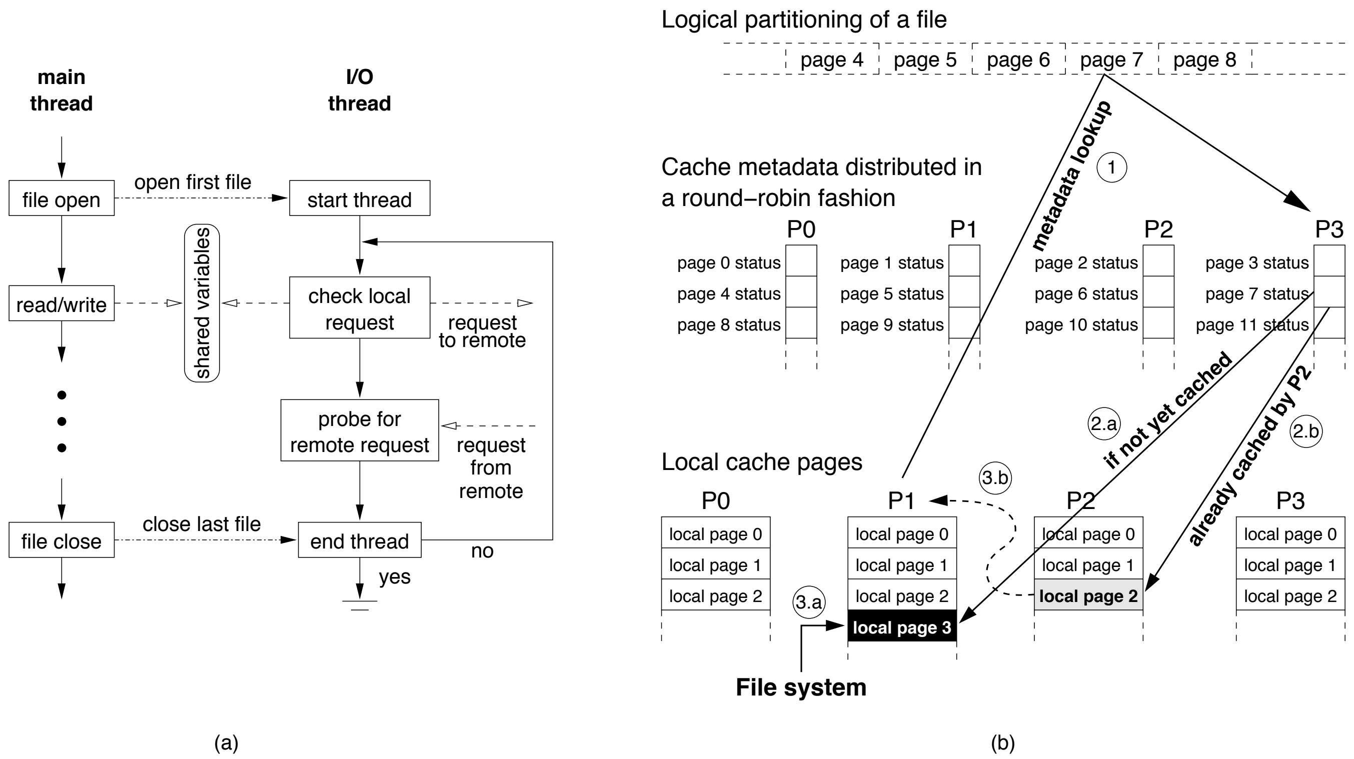

Figure 1: (a) The I/O thread operations from a single MPI process' view. (b) Example of the I/O flow in MPI-IO caching where MPI process P_1 reads data from logical file page 7.

the caching system needs a transparent mechanism in each process that can run independently and concurrently with the main program. We create a thread in each MPI process when the first file is opened and keep the threads alive until the last file is closed. The I/O thread carries out the work of file I/O and caching without interrupting the main thread. Figure 1(a) illustrates the I/O thread's operations from a single MPI process's point of view. This approach is particularly important for MPI applications that make independent I/O calls. Since independent I/O may be initiated by each process in an unrelated manner and requires no process synchronization, any collaboration among the processes' main threads is difficult. In our design, each I/O thread handles both local and remote requests to the locally cached data, and cooperates with remote threads for coherence control. The I/O thread communicates with the main thread through a POSIX mutex protected shared variable and uses `MPI_Iprobe()` to detect remote requests. Blocking MPI communication functions such as `MPI_Wait()` and `MPI_Probe()` cannot be used, because the I/O thread must be always available to respond to requests for any of the opened files. In fact, our implementation uses only asynchronous MPI communication calls for this very reason.

3.1 Cache Metadata Management

Our caching scheme logically divides a file into equally sized pages, each of which can be cached. The default cache page size is set to the file system block size and is also adjustable through an MPI hint. A page size aligned with the file system lock granularity is recommended, since it prevents false sharing. Cache metadata describing the cache status of these pages are statically distributed in a round-robin fashion among the MPI processes that collectively open the shared file. Finding the MPI rank of the process storing a page's metadata requires only a modulus operation. This distributed approach avoids centralization of metadata management. Cache metadata includes the MPI rank of the page's current location, lock mode, and the page's recent access history. The lock mode is separated into sharable read locks and exclusive write locks. A page's access history is used for cache eviction and page migration policies.

To ensure cache metadata integrity (atomic access to metadata), a distributed locking mechanism is implemented where each MPI process manages the lock requests for its assigned metadata. Metadata locks must be obtained before an MPI process can freely access the metadata, cache page, and the file range corresponding to the page. If an I/O request covers multiple consecutive file blocks that are currently cached at different MPI processes, all cache pages must be locked prior to their access. Since dead lock may occur when more than two processes are simultaneously requesting locks for the same two pages, we employ the two-phase locking strategy proposed in [1]. Under this strategy, lock requests are issued in a strictly increasing page ID order and the prior page lock must be obtained before requesting the next lock.

3.2 Cache Page Management

To simplify coherence control, we allow at most a single cached copy of file data among all MPI processes. When accessing a file page that is not being cached anywhere, the requesting process will try to cache the page locally, by reading the entire page from the file system if it is a read operation, or by reading the necessary part of the page if it is a write operation. An upper bound, by default 32 MB, indicates the maximum memory size that can be used for caching. If the memory allocation utility, `malloc()` finds enough memory to accommodate the page and the total allocated cache size is below the upper bound, the page will be cached. Otherwise, under memory pressure, the page eviction routine is activated. Eviction is solely based on only local references and a least-recent-used policy. If the requested file pages are not cached and the request amount is larger than the upper bound, the read/write calls will go directly to the file system. If the requested page is already cached locally, a simple memory copy will satisfy the request. If the page is cached at a remote process, the request is forwarded to the page owner. An example I/O flow for a read operation is illustrated in Figure 1(b) with four MPI processes. In this example, process P_1 reads data in file page 7. The first step is to lock and retrieve the metadata of page 7 from P_3 (7 mod 4 = 3). If the page is not cached yet, P_1 will cache it locally (into local page 3) by reading from the file system, as depicted by steps (2.a) and (3.a). If the metadata indicates that the page is currently cached on P_2, then an MPI message is sent from P_1 to P_2 asking for data transfer. In step (3.b), assuming file page 7 is cached in local page 2, P_2 sends the requested data to P_1.

When closing a file, all dirty cache pages are flushed to the file system. A high water mark is used in each cache page to indicating the range of dirty data, so that flushing needs not always be an entire page. Because logically contiguous file pages are potentially scattered across MPI processes, a two-phase flushing is devised at file close to shuffle cache pages such that neighboring cache pages are moved to the same processes before the flush. Although shuffling requires extra communication cost, this approach enables sequential file access and further improves the performance.

4. TWO-STAGE WRITE-BEHIND BUFFERING

We used our MPI-IO caching design as a basis for constructing a two-stage write-behind method to focus on improving the performance of write-only operations. While MPI-IO caching can handle any I/O operations, this write-behind method has two restrictions: the file must be opened in write-only mode and the MPI atomic mode must be set to false. The former is indicated by the use of `MPI_MODE_WRONLY` in the access mode argument of `MPI_File_open()`. Note that the MPI atomic mode is set to false by default but is changeable through function `MPI_File_set_atomicity()`. In the first stage, write data is accumulated in local buffers along with the requesting file offset and length. Once the local buffer is full, the data is sent to the global buffer at the second stage. Double buffering is used so one buffer can accumulate incoming write data while the other is asynchronous sent to the global buffer. The global buffer's assignment consists of file pages statically distributed among the MPI processes that collectively open a shared file. Similar to IBM's data shipping approach, a file is logically divided into equally sized pages with each page bound to a single MPI process in a round-robin striping scheme. Accordingly, page i resides on the process of rank (i mod *nproc*), where *nproc* is the number of processes. To minimize lock contention in the underlying file system, the default page size is set to the file system stripe size, but it is adjustable through an MPI hint.

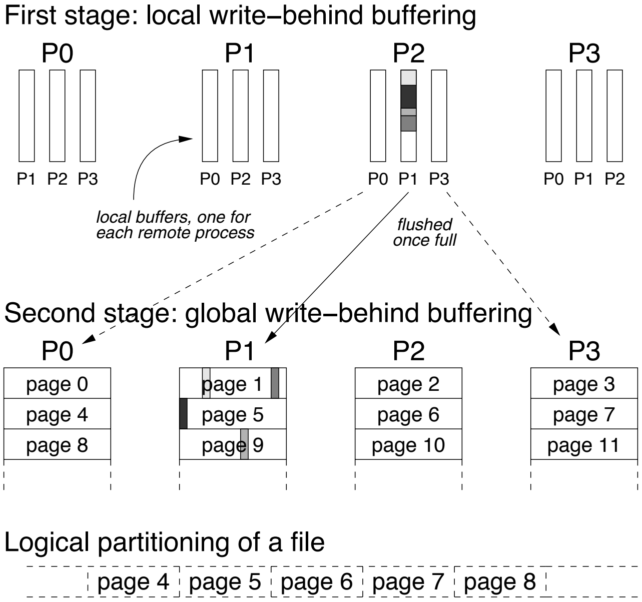

Figure 2: Design of two-stage write-behind buffering.

The first-stage local buffer is separated into (*nproc* - 1) sub-buffers: each of which is dedicated to a remote MPI process. When accumulating in the local buffer, write data is appended to the sub-buffer corresponding to its destination MPI process. Consequently, data for a write request spanning more than one page in the global buffer space will be split and appended to different local buffers. The default size for each local sub-buffer is 64 KB, but can be changed by the application through an MPI hint. When flushing the local buffers to the global buffers, the supplemental offset and length information for each request is sent along with the data. Since a local sub-buffer may contain data spanning multiple global pages on the same destination process, the destination process must use the offset-length information to correctly distribute the incoming data to the proper pages. Global file pages reside in memory until their eviction is necessary. Figure 2 illustrates the operations of the two-stage write-behind method. In this example, there are four MPI processes and each has three first-stage local sub-buffers. Each sub-buffer is corresponding to a remote process. At P_2, when the sub-buffer tied to P_1 is full, it is flushed to P_1. After receiving the flushed data from P_2, P_1 redistributes it to the global pages based on the included offset-length information.

Since the global buffer is distributed over multiple processes, each process must be able to respond to remote requests for flushing the first-stage write-behind data. Similar to the caching implementation, we use the I/O thread approach to permit the process collaboration without interrupting the main program thread. Each process can have multiple files opened, but only one thread is created. Our algorithm creates the I/O threads when the program opens its first file and terminates the I/O thread when all files are closed. Once an I/O thread is created, it enters an infinite loop to serve both local and remote write requests until it is signaled to terminate by the main thread. All operations related to I/O and data buffering are carried out by the I/O thread alone. A shared conditional variable protected by a mutex is used to indicate whether a write request has been issued by the main thread or the I/O thread has completed a request. Other communication between the main thread and the I/O thread is also done through a few shared variables which contain information such as the file handler, write offset, write buffer, etc. To serve remote requests, the I/O thread probes for incoming I/O requests from all processes in the MPI communicator.

5. EXPERIMENTAL RESULTS

Our implementations for MPI-IO caching and two-stage write-behind are evaluated on two machines: Tungsten and Mercury, at the National Center for Supercomputing Applications. Tungsten is a 1280-node Dell Linux cluster where each node contains two Intel 3.2 GHz Xeon processors sharing 3 GB of memory. The compute nodes run a Red Hat Linux operating system and are inter-connected by both Myrinet and Gigabit Ethernet communication networks. A Lustre parallel file system version 1.4.4.5 is installed on Tungsten. The lock granularity of Lustre is the system page size: 4 KB on Tungsten. Output files are saved in a directory configured with a stripe count of 16 and a 512 KB stripe size. All files created in this directory share the same striping parameters. Mercury is an 887-node IBM Linux cluster where each node contains two Intel 1.3/1.5 GHz Itanium II processors sharing 4 GB of memory. Running a SuSE Linux operating system, the compute nodes are inter-connected by both Myrinet and Gigabit Ethernet. Mercury runs an IBM GPFS parallel file system version 3.1.0 configured in the Network Shared Disk (NSD) server model with 54 I/O servers and 512 KB file block size. The lock granularity on GPFS is the disk sector size, 512 bytes on Mercury. Note that because IBM's MPI library is not available on Mercury, we could not comparatively evaluate the performance of GPFS's data shipping mode. MPI-IO caching is implemented in the ROMIO layer of MPICH version 2-1.0.5, the latest thread-safe version of MPICH2 at the time our experiments were performed. Support for thread-safety is limited, however, to the default sock channel of MPICH2; thereby restricting inter-process communication in our experiments to the slower Gigabit Ethernet.

In our experiments, we use a 512 KB page size for both MPI-IO caching and two-stage write-behind method. Setting the cache page size to the file system stripe size aligns all write requests to the stripe boundaries and hence lock boundaries. For performance evaluation, we use three benchmarks: BTIO, FLASH I/O, and S3D I/O. We report the aggregate write bandwidth, because the asynchronous MPI functions used in our implementation prevent accurate breakdowns of the costs for computation, communication, and file I/O. The I/O bandwidth numbers are obtained by dividing the aggregate write amount by the time measured from the beginning of file open until after file close. Note that although no explicit file synchronization is called in these benchmarks, closing files will flush all dirty data in both MPI-IO caching and the two-stage write-behind implementations to the file system.

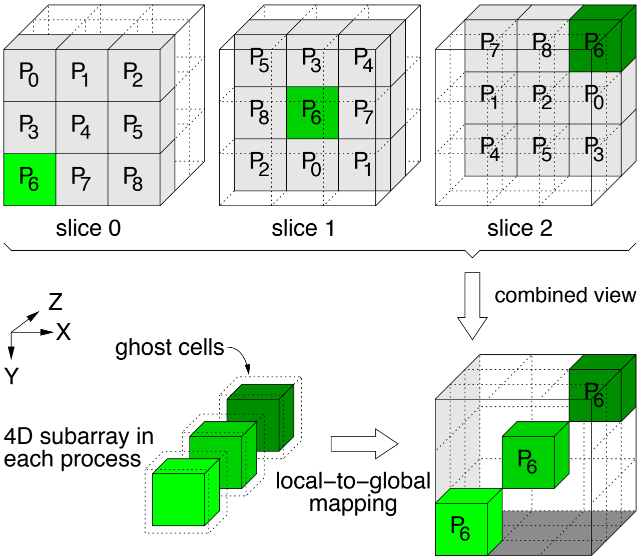

Figure 3: BTIO data partitioning pattern. The 4D subarray in each process is mapped to the global array in a block-tridiagonal fashion. This example uses 9 processes and highlights the mapping for process P_6.

5.1 BTIO Benchmark

Developed by NASA Advanced Supercomputing Division, the parallel benchmark suite NPB-MPI version 2.4 I/O is formerly known as the BTIO benchmark [28]. BTIO presents a block-tridiagonal partitioning pattern on a three-dimensional array across a square number of processes. Each process is responsible for multiple Cartesian subsets of the entire data set, whose number increases with the square root of the number of processors participating in the computation. Figure 3 illustrates the BTIO partitioning pattern with an example of nine processes. BTIO provides options for using either MPI collective or independent I/O. In BTIO, forty arrays are consecutively written to a shared file by appending one after another. Each array must be written in a canonical, row-major format in the file. We evaluate the Class C data size with array dimensionality 162 × 162 × 162 and an aggregate write amount for a complete run of 6.34 GB. With this fixed aggregate write amount, we evaluate BTIO using different number of MPI processes; therefore, the amount written by each process decreases as the number of processes increases.

BTIO performance results are given in Figure 4. It shows write bandwidths for three scenarios: using collective MPI I/O functions natively, using collective I/Os with MPI-IO caching, and using independent I/Os with the two-stage write-behind method. Our experiments verify the performance evaluation reported in [5], using independent I/O natively results in significantly worse write bandwidth than using collective I/O. This is because there is a significant difference in the number of the write requests to the file system. Table 1 shows the number of file system `write()` calls and their data amounts generated at the ADIO layer. Since

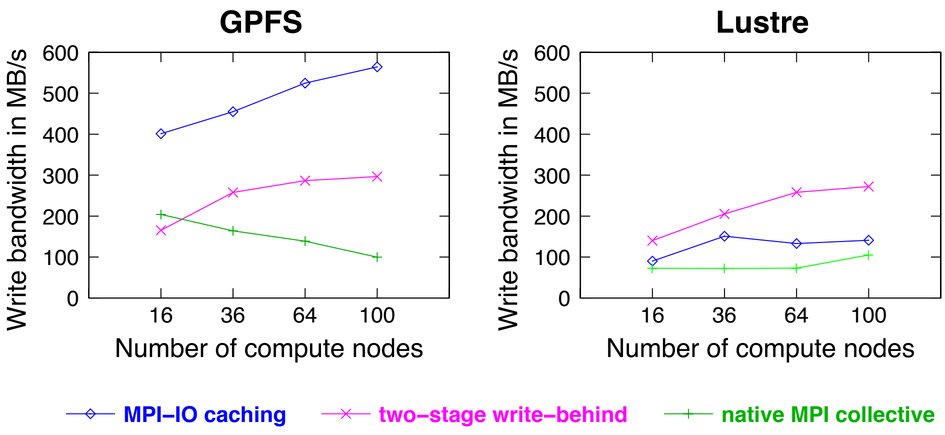

Figure 4: Performance results of BTIO benchmark.

ROMIO uses the two-phase I/O strategy in its collective I/O implementation to redistribute and aggregate I/O requests to large contiguous file accesses, the number of write requests generated by collective I/O is significantly less than independent I/Os. The extremely large number of write requests generated by native independent I/O in BTIO makes the aggregate write performance very poor. With less than 5 MB per second bandwidth in our experience, the results for independent I/O alone are not presented.

On the GPFS and Lustre file systems, both the MPI-IO caching and write-behind methods outperform the native MPI collective I/O. The performance improvement is attributed to reduced file system lock contention. For each collective write operation in BTIO, the aggregate write amount is 162.18 MB. When partitioned evenly among all MPI processes, the file domains are not aligned with the file system lock boundaries. Table 1 also shows the write amounts per request per process when collective MPI I/O is used. It is clear that they do not generate aligned request offsets for all four numbers of processes used in our experiments. Thus, conflict locks due to false sharing occur and hence serialize the concurrent write requests. As for the two-stage write-behind method, there is a significant improvement over not only the native independent I/O, but also the native collective I/O. This improvement is attributed to the local data accumulation at the first stage write-behind achieving better network bandwidth as well as the second-stage write request alignment reducing file system lock contention. On Lustre, the write-behind method even outperforms the MPI-IO caching. In fact, both the MPI-IO caching and two-stage write-behind methods have their own advantages and disadvantages. As we described earlier, the distributed lock protocol used in MPI-IO caching to maintain the cache metadata integrity incurs a certain degree of communication overhead. Because the write-behind method deals with write-only patterns, no coherence control is required at all. MPI-IO caching allows the first process that requests a page to buffer it locally; thereby, taking advantage of data locality and increasing the likelihood of local cache hits. In contrast, since the second-stage buffers of the write-behind method are statically assigned across MPI processes in a round-robin fashion, the data written by a process in the first-stage buffers will most likely need to be flushed to remote processes.

Table 1: Number of write requests and data amount generated by the BTIO benchmark when running MPI I/O natively.

Number of	Collective I/O		Independent I/O	
processes	NWRPP	WAPRPP	NWRPP	WAPRPP
16	40	10.14 MB	262440	1620 B
36	40	4.51 MB	174960	1080 B
64	40	2.53 MB	131240	810 B
100	40	1.27 MB	105000	405 B

NWRPP: number of write requests per process
WAPRPP: write amount per request per process

5.2 FLASH I/O Benchmark

The FLASH I/O benchmark suite [29] is the I/O kernel of the FLASH application, a block-structured adaptive mesh hydrodynamics code that solves fully compressible, reactive hydrodynamic equations, developed mainly for the study of nuclear flashes on neutron stars and white dwarfs [6]. The computational domain is divided into blocks that are distributed across a number of MPI processes. A block is a three-dimensional array with an additional 4 elements as guard cells in each dimension on both sides to hold information from its neighbors. In our experiments, we used a 16 × 16 × 16 block size that produces about 2.5 MB of write data per process in each MPI I/O operation. There are 24 variables per array element, and about 80 blocks on each MPI process. A variation in block numbers per MPI process is used to generate a slightly unbalanced I/O load. Since the number of blocks is fixed for each process, increase in the number of MPI processes linearly increases the aggregate I/O amount as well. FLASH I/O produces a checkpoint file and two visualization files containing centered and corner data. The largest file is the checkpoint, the I/O time of which dominates the entire benchmark. FLASH I/O uses the HDF5 I/O interface to save data along with its metadata in the HDF5 file format. Since the implementation of HDF5 parallel I/O is built on top of MPI-IO [7], the MPI performance effects of the MPI-IO caching and two-stage write-behind methods can be observed in overall FLASH I/O performance. To eliminate the overhead of memory

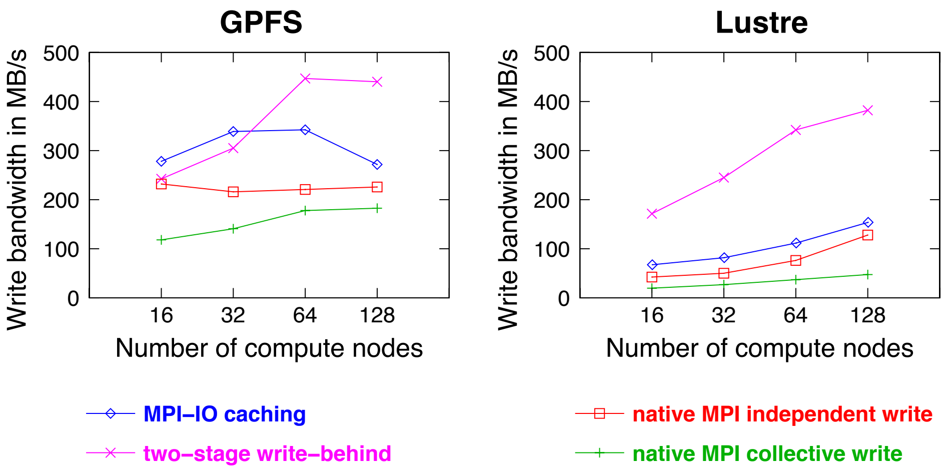

Figure 5: Performance results of FLASH I/O benchmark.

copying in the HDF5 hyper-slab selection, FLASH I/O extracts the interiors of the blocks via a direct memory copy into a buffer before calling the HDF5 functions. There are 24 I/O loops, one for each of the 24 variables. In each loop, every MPI process writes into a contiguous file space, appending its data to the previous ranked MPI process; therefore, a write request from one process does not overlap or interleave with the request from another. In ROMIO, this non-interleaved access pattern actually triggers the independent I/O subroutines, instead of collective subroutines, even if MPI collective writes are explicitly called. This behavior can be overridden by enabling the `romio_cb_write` hint. For the FLASH I/O pattern, forcing collective I/O creates a balanced workload, but adds extra communication costs.

Figure 5 shows the performance results of the FLASH I/O benchmark. The aggregate write bandwidth is calculated by dividing the data size written to all three files by the overall execution time. At first, we observe that the results of forcing collective writes are worse than using the default independent writes. This is because the cost of rebalancing the I/O load overwhelms the benefit of a balanced I/O. From this we can conclude that whether or not the I/O load is balanced is not a significant enough factor in the I/O performance of FLASH. Both the MPI-IO caching and two-stage write-behind methods outperform the native independent writes. Since the data amount per write request in each MPI process is about 2.5 MB, larger than the system stripe size of 512 KB, the improvement is due to the fact that accumulating write data in both caching and write-behind enables write request alignment to the file system stripe boundaries. Similar to our analysis for BTIO, such alignment significantly eliminates the lock contention that would otherwise occur in the underlying GPFS and Lustre file systems from the use of native MPI I/O. Note that FLASH I/O writes both array data and metadata through the HDF5 I/O interface to the same file. Metadata, usually stored at the file header, may cause unaligned write requests for array data when using native MPI I/O.

5.3 S3D I/O Benchmark

The S3D I/O benchmark is the I/O kernel of the S3D application, a parallel turbulent combustion application using a direct numerical simulation solver developed at Sandia National Laboratories [20]. S3D solves fully compressible Navier-Stokes, total energy, species and mass continuity equations coupled with detailed chemistry. The governing equations are solved on a conventional three-dimensional structured Cartesian mesh. A checkpoint is performed at regular intervals, and its data consists primarily of the solved variables in 8-byte three-dimensional arrays, corresponding to the values at the three-dimensional Cartesian mesh points. During the analysis phase the checkpoint data can be used to obtain several more derived physical quantities of interest; therefore, a majority of the checkpoint data is retained for later analysis. At each checkpoint, four global arrays are written to files and they represent the variables of mass, velocity, pressure, and temperature, respectively. Mass and velocity are four-dimensional arrays while pressure and temperature are three-dimensional arrays. All four arrays share the same size for the lowest three spatial dimensions X, Y, and Z, and they are all partitioned among MPI processes along X-Y-Z dimensions in the same block-block-block fashion. For the mass and velocity arrays, the length of the fourth dimension is 11 and 3, respectively. The fourth dimension is not partitioned.

In the original S3D, the I/O is programmed in Fortran I/O functions and each process writes all its sub-arrays to a new, separate file at each checkpoint. We added the functionality of using MPI I/O to write the arrays into a shared file in their canonical order. With this change, there is only one file created per checkpoint, regardless of the number of MPI processes used. Figure 6 shows the data partitioning pattern on a 3D array and the mapping of a 4D sub-array to the global array in file. 3D arrays can be considered a special case with a fourth dimension length of one. For performance evaluation, we keep the size of partitioned X-Y-Z dimensions a constant $50 \times 50 \times 50$ in each process. This produces about 15.26 MB of write data per process per checkpoint. Similar to FLASH I/O, as we increase the number of MPI processes, the aggregate I/O amount proportionally increases as well. We measure the execution time for running ten checkpoints. Figure 7 shows the write bandwidth results for using For-

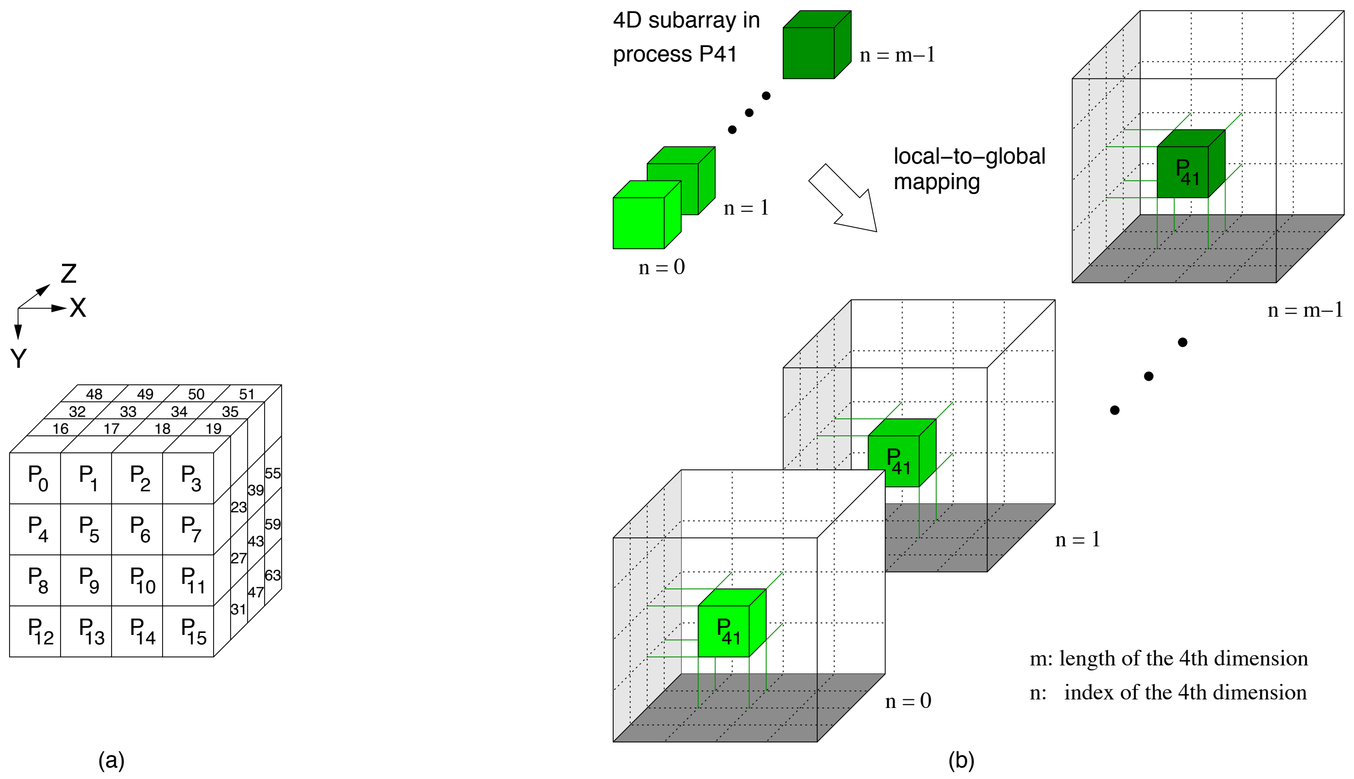

Figure 6: S3D I/O data partitioning pattern. (a) For 3D arrays, the sub-array of each process is mapped to the global array in a fashion of block partitioning in all X-Y-Z dimensions. (b) For 4D arrays, the lowest X-Y-Z dimensions are partitioned the same as the 3D arrays while the fourth dimension is not partitioned. This example uses 64 processes and highlights the mapping of process P_{41}'s sub-array to the global array.

tran I/O, MPI collective I/O natively, collective I/O with MPI-IO caching, and independent I/O with the two-stage write-behind method. Fortran I/O has significantly better performance than the other cases on Lustre, but this situation changes on GPFS. On GPFS, Fortran I/O is even worse than the MPI-IO caching for 64 and 128 processes. Further investigation by separately measuring the file open time of ten checkpoints shows that file open costs increase more dramatically on GPFS than Lustre when scaling the number of processes. Lustre seems to handle larger number of files more efficiently than GPFS. As for the MPI I/O performance, we observe that MPI-IO caching outperforms the native collective I/O on both GPFS and Lustre. Similar to BTIO and FLASH I/O, the effect of I/O alignment enhances the performance of MPI-IO caching. The two-stage write-behind behaves differently on each file system in that it is worse than the native collective I/O on GPFS, but outperforms the MPI-IO caching on Lustre. We think write-behind performance is worse because the S3D I/O pattern results in many remote data flushes from the local first-stage to global second-stage buffers. Note that the native MPI I/O case uses collective I/O functions, while the write-behind method uses independent I/O functions. Compared to the use of native independent writes that produces I/O bandwidth less than 5 MB per second, our write-behind method shows a clear improvement and achieves a bandwidth close to, or better than, the collective I/O method.

6. CONCLUSIONS

Client-side file caching has been demonstrated as a powerful tool for file systems to reduce the data transfer between clients and the I/O servers. If not used carefully, however, caching can seriously hurt the shared-file parallel I/O performance. As discussed in this paper, there are several non-obvious factors that influence the performance, such as false sharing and I/O atomicity. Besides lock contention, other costs for file caching include operations for extra memory copying, and memory space management for cache data. When this combined overhead overwhelms caching benefits, caching only reduces performance. Another factor also significantly affecting a caching system's performance is the available memory space. The performance evaluation of a caching system is different from measuring the maximum data rate for a file system. Typical file system benchmarks avoid caching effects by using an I/O amount larger than the aggregated memory size of either clients or servers. In contrast, file caching performance can really only be captured when there is sufficient unused memory space for the caching system to operate in. All these factors contribute to the complex challenge of evaluating and explaining the parallel I/O performance of a caching system in great detail.

We use the BTIO benchmark and two production application's I/O kernels, FLASH I/O and S3D I/O, to evaluate our idea of a file caching layer between the applications and file systems. In the real world, applications' parallel I/O patterns may not always result in the aligned I/O requests used in many I/O benchmark suites. Our experiments demon-

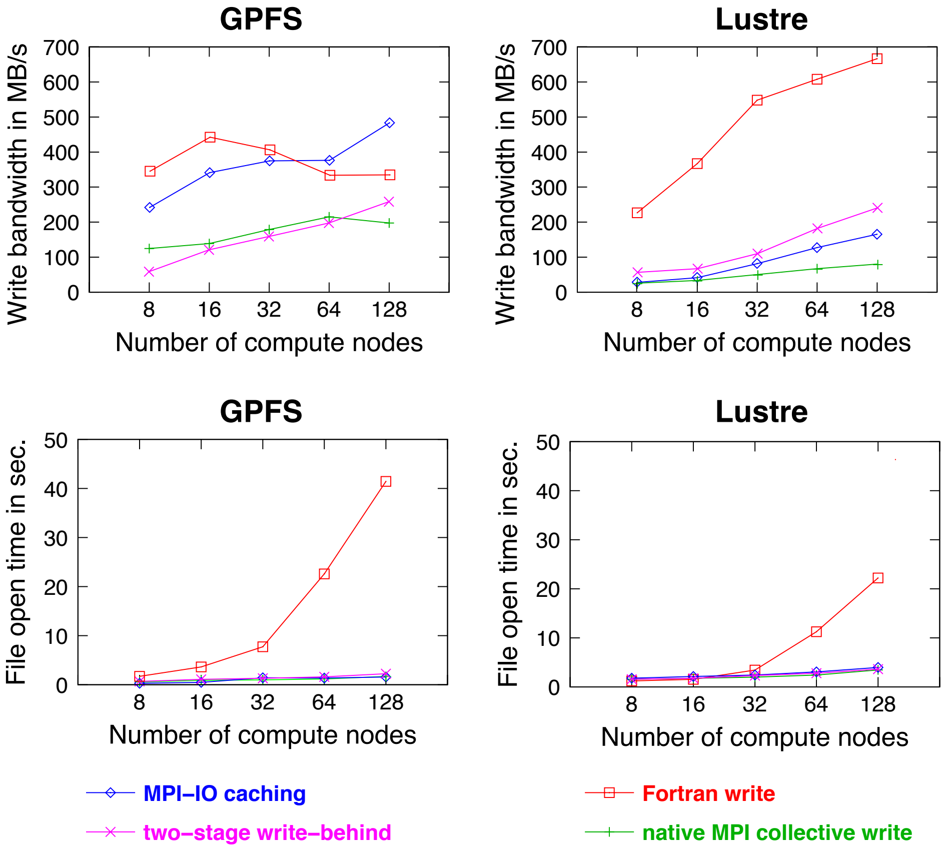

Figure 7: Write bandwidth and file open timing for S3D I/O benchmark.

strate the significance of aligning the I/O requests with the file system lock boundaries. In fact, enforcing POSIX semantics such as I/O atomicity has become substantial obstacles to parallel file systems efficiently handling shared-file I/O at sustained I/O rates close to the maximum network bandwidth. Since the primary I/O patterns of large-scale parallel applications are write-only and non-overlapping, enforcing strict semantics like POSIX atomicity is not always necessary. We plan to add a mechanism in MPI-IO caching to selectively relax the atomicity requirement. Our current design for MPI-IO caching conservatively chooses to keep at most a single copy of cached file data. Although this policy simplifies coherence control, it also increase the likelihood of accessing remote cache pages instead of local cache pages. We will investigate a new implementation that allows multiple cached copies of the same file date. We also plan to explore other issues such as cache load re-balancing and data prefetching.

7. ACKNOWLEDGMENTS

This work was supported in part DOE's SCiDAC program (Scientific Data Management Center), award number DE-FC02-07ER25808, NSF/DARPA ST-HEC program under grant CCF-0444405, NSF HECURA CCF-0621443 and NSF NGS program under grant CNS-0406341. We acknowledge the use of the Dell Xeon Cluster and IBM IA-64 Linux Cluster at the National Center for Supercomputing Applications under TeraGrid Project TG-CCR060017T.

8. REFERENCES

[1] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[2] T. Corts, S. Girona, and J. Labarta. PACA: A Cooperative File System Cache for Parallel Machines. In *the 2nd International Euro-Par Conference*, pages 477–486, August 1996.

[3] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *the First Symposium on Operating System Design and Implementation*, November 1994.

[4] J. del Rosario, R. Brodawekar, and A. Choudhary. Improved Parallel I/O via a Two-Phase Run-time Access Strategy. In *the Workshop on I/O in Parallel Computer Systems at IPPS '93*, pages 56–70, April 1993.

[5] S. Fineberg, P. Wong, B. Nitzberg, and C. Kuszmaul. PMPIO - A Portable Implementation of MPI-IO. In *the 6th Symposium on the Frontiers of Massively Parallel Computation*, October 1996.

[6] 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*, pages 131–273, 2000.

[7] HDF Group. *Hierarchical Data Format, Version 5.* The National Center for Supercomputing Applications, http://hdf.ncsa.uiuc.edu/HDF5.

[8] J. Huber, C. Elford, D. Reed, A. Chien, and D. Blumenthal. PPFS: A High Performance Portable File System. In *the 9th ACM International Conference on Supercomputing*, 1995.

[9] IEEE/ANSI Std. 1003.1. *Portable Operating System Interface (POSIX)-Part 1: System Application Program Interface (API) [C Language]*, 1996.

[10] 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 *the 18th annual international conference on Supercomputing*, pages 58–67, June 2004.

[11] W. Liao, A. Ching, K. Coloma, A. Choudhary, and L. Ward. An Implementation and Evaluation of Client-Side File Caching for MPI-IO. In *the International Parallel and Distributed Processing Symposium (IPDPS)*, March 2007.

[12] W. Liao, K. Coloma, A. Choudhary, L. Ward, E. Russell, and S. Tideman. Collective Caching: Application-aware Client-side File Caching. In *the 14th IEEE International Symposium on High Performance Distributed Computing (HPDC-14)*, July 2005.

[13] Lustre: A Scalable, High-Performance File System. *Whitepaper.* Cluster File Systems, Inc., 2003.

[14] X. Ma, M. Winslett, J. Lee, and S. Yu. Improving MPI-IO Output Performance with Active Buffering Plus Threads. In *the International Parallel and Distributed Processing Symposium (IPDPS)*, April 2003.

[15] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard, Version 1.1*, June 1995. http://www.mpi-forum.org/docs/docs.html.

[16] Message Passing Interface Forum. *MPI-2: Extensions to the Message Passing Interface*, July 1997. http://www.mpi-forum.org/docs/docs.html.

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

[18] J. Prost, R. Treumann, R. Hedges, A. Koniges, and A. White. Towards a High-Performance Implementation of MPI-IO on top of GPFS. In *the Sixth International Euro-Par Conference on Parallel Processing*, August 2000.

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

[20] R. Sankaran, E. Hawkes, J. Chen, T. Lu, and C. Law. Direct Numerical Simulations of Turbulent Lean Premixed Combustion. *Journal of Physics: conference series*, 46:38–42, 2006.

[21] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *the Conference on File and Storage Technologies (FAST'02)*, pages 231–244, January 2002.

[22] A. Tanenbaum and M. van Steen. *Distributed Systems - Principles and Paradigms.* Prentice Hall, 2002.

[23] R. Thakur and A. Choudhary. An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays. *Scientific Programming*, 5(4):301–317, 1996.

[24] R. Thakur, W. Gropp, and E. Lusk. An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. In *the 6th Symposium on the Frontiers of Massively Parallel Computation*, October 1996.

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

[26] R. Thakur, W. Gropp, and E. Lusk. On Implementing MPI-IO Portably and with High Performance. In *the Sixth Workshop on I/O in Parallel and Distributed Systems*, pages 23–32, May 1999.

[27] G. Voelker, E. Anderson, T. Kimbrel, M. Feeley, J. Chase, A. Karlin, and H. Levy. Implementing Cooperative Prefetching and Caching in a Globally-managed Memory System. In *the Joint International Conference on Measurement and Modeling of Computing Systems*, pages 33–43, 1998.

[28] P. Wong and R. der Wijngaart. NAS Parallel Benchmarks I/O Version 2.4. Technical Report NAS-03-002, NASA Ames Research Center, Moffet Field, CA, January 2003.

[29] M. Zingale. FLASH I/O Benchmark Routine – Parallel HDF 5, March 2001. http://flash.uchicago.edu/∼zingale/ flash_benchmark_io.