An Implementation and Evaluation of Client-Side File Caching for MPI-IO

Wei-keng Liao[1], Avery Ching[1], Kenin Coloma[1], Alok Choudhary[1], and Lee Ward[2]

[1]Northwestern University
Dept. of Electrical Engineering and Computer Science
Evanston, IL 60208, USA
{wkliao,aching,kcoloma,choudhar}@ece.northwestern.edu

[2]Sandia National Laboratories
Dept. of Scalable Computing Systems
Albuquerque, NM 87185, USA
lee@sandia.gov

Abstract

Client-side file caching has long been recognized as a file system enhancement to reduce the amount of data transfer between application processes and I/O servers. However, caching also introduces cache coherence problems when a file is simultaneously accessed by multiple processes. Existing coherence controls tend to treat the client processes independently and ignore the aggregate I/O access pattern. This causes a serious performance degradation for parallel I/O applications. In our earlier work, we proposed a caching system that enables cooperation among application processes in performing client-side file caching. The caching system has since been integrated into the MPI-IO library. In this paper we discuss our new implementation and present an extended performance evaluation on GPFS and Lustre parallel file systems. In addition to comparing our methods to traditional approaches, we examine the performance of MPI-IO caching under direct I/O mode to bypass the underlying file system cache. We also investigate the performance impact of two file domain partitioning methods to MPI collective I/O operations: one which creates a balanced workload and the other which aligns accesses to the file system stripe size. In our experiments, alignment results in better performance by reducing file lock contention. When the cache page size is set to a multiple of the stripe size, MPI-IO caching inherits the same advantage and produces significantly improved I/O bandwidth.

1. Introduction

Since the 1990s, parallel file systems have been built based on the experience learned from distributed file systems and focused on providing high data throughput and scalability. Many performance enhancement strategies developed for distributed environments have also been adopted. One example is client-side file caching, which aims to reduce the number of data transfers between file servers and application clients. However, a cache coherence problem is introduced when changes to a local copy of cached data do not propagate to other copies in a timely manner, leaving the cache in an incoherent state. Coherence control used in distributed file systems is often implemented by the bookkeeping of cache status at the servers and invoking client callbacks when flushing dirty cache is needed. This traditional approach handles I/O requests without regard to the correlation among requesting clients. While this assumption may work well in the distributed environment, it can be inefficient for parallel applications in which multiple clients often work on globally partitioned data structures and concurrently read/write these data structures from/to shared files.

In our earlier work, we prototyped a user-level caching system by incorporating the MPI communicator concept to enable cooperative caching among MPI processes that open a shared file collectively [13]. By moving the caching layer closer to user applications, we believe that high-level application I/O patterns can be utilized for better coherence control. In particular, knowing the group of clients that will later access a shared file can help track incoherent cache status effectively. We use an I/O thread in each application process to handle local file caching and remote cache page access. All I/O threads cooperate with each other for cache coherence control. One immediate benefit of this design is that the file system can pass consistency control responsibilities to the caching system.

We have since expanded the functionality of the caching system and integrated it into ROMIO, an MPI-IO implementation developed at Argonne National Laboratory [23]. In the rest of the paper, we refer to our new implementation as MPI-IO caching. New functionality includes the following: automatically increase the memory space for cache metadata as the file size grows; separate sharable

1-4244-0910-1/07/$20.00

read locks from exclusive write locks; pipelined two-phase locking that enables overlapping of cache page access and lock requests; a cache page migration mechanism; dynamic cache page allocation based on available memory space; a two-phase flushing mechanism at file close to shuffle the cache pages such that neighboring pages are moved to the same processes; handling multiple files opened with different MPI communicators; MPI hints to enable and setting caching parameters, such as a customized cache page size, enabling/disabling page migration, pipelined locking, and two-phase flushing.

In general, client-side file caching enhances I/O performance under two scenarios: the I/O pattern with repeated accesses to the same file regions, and the pattern with a large number of small requests. For the former, caching reduces client-server communication costs. In this paper, we will not discuss this pattern in detail, since it is not commonly seen in today's parallel applications. As for the latter, caching accumulates multiple small requests into large requests for better network utilization, also known as write behind and read ahead. However, caching introduces overheads of coherence control, operations for extra memory copying, and memory space management. When such overheads overwhelm caching benefits, enabling caching only results in reducing performance. In this work we use BTIO and FLASH I/O benchmarks to present a comprehensive performance evaluation for MPI-IO caching. Our discussion focuses on the analysis of the performance impact from several system features, such as distributed file locking, direct I/O, and file domain alignment to the file system stripe size. The performance results demonstrate that MPI-IO caching succeeds in using a write behind strategy to align I/O requests with the file system block size, which effectively reduces lock contention that otherwise appears in non-aligned accesses. The rest of the paper is organized as follows. Section 2 discusses related works. Section 3 describes the design of MPI-IO caching and its recent improvement. Experimental results and performance analysis are given in Section 4. The paper is concluded in Section 5.

2. Related Work

Client-side file caching is supported in many parallel file systems, for instance, IBM GPFS [18, 21] and Lustre [14]. GPFS, by default, employs a distributed locking mechanism to maintain a coherent cache between nodes, in which lock tokens must be granted before any I/O operation can be performed [19]. Distributed locking avoids centralized lock management by making a token holder a local lock manager for granting any further lock requests to its granted byte range. A token allows a node to cache data that cannot be modified elsewhere without revoking the token first. IBM's MPI-IO implementation over GPFS uses a different mechanism named data shipping, where a file is divided into equal sized blocks, each bound to a single I/O agent, a thread in an MPI process. A file block can only be cached by its I/O agent which is responsible for all accesses to this block. I/O operations must go through the I/O agents which "ship" the requested data to the appropriate processes. Data shipping avoids the cache coherence problem by allowing at most one copy of file data to be cached among agents. The Lustre file system uses a slightly different distributed locking protocol in which each I/O server manages locks for the stripes of file data it owns. If a client requests a conflict lock held by another client, a message is sent to the lock holder asking for the lock to be released. Before a lock can be released, dirty cache must be flushed to the servers.

Cooperative caching has been proposed as a system-wise solution for caching and coherence control [3]. It coordinates multiple clients by relaying the requests not satisfied by one client's local cache to another client. Systems that use cooperative caching include PGMS [24], 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 [11]. However, cooperative caching in general requires changes in the file system at both client and server. In contrast, the MPI-IO caching proposed in this work is implemented in user space and requires no change to the underlying file system.

2.1. MPI-IO

The Message Passing Interface standard (MPI) defines a set of programming interfaces for parallel program development that explicitly uses message passing to carry out inter-process communication [16]. The MPI standard version 2 extends the interfaces to include file I/O operations [17]. MPI-IO inherits two important MPI features: the ability to define a set of processes for group operations using an MPI communicator and the ability to describe complex memory layouts using MPI derived data types. A communicator specifies the processes that participate in an MPI operation, whether for inter-process communication or I/O requests to a shared file. For file operations, an MPI communicator is required when opening a file in parallel to indicate the processes that will later access the file. Two types of functions are defined in MPI-IO, namely collectives and independents. Collective operations require all the processes that opened the file to participate. Many collective I/O optimizations take advantage of the synchronization by having processes exchange individual access patterns in order to produce a better I/O strategy. A well-known example of a collective I/O optimization is two-phase I/O [4]. In contrast, independent I/O does not require synchronization, making any cooperative optimizations very difficult.

Active buffering is considered an enhancement for MPI

collective write operations [15]. It buffers write data locally and uses an I/O thread to flush the accumulated data to the file system in the background. Using I/O threads allows dynamic adjustment to the local buffer size based on the available memory space. When processing a write request, the main thread duplicates the writing data to a newly allocated buffer and appends this buffer into a queue. The I/O thread later retrieves the buffers from the head of the queue, makes write calls to the file system, and releases the buffer memory. Active buffering aims to improve performance for applications that make only collective write calls. Lacking consistency control, it is hard for active buffering to handle the I/O with mixed reads and writes, as well as independent and collective calls.

3. Client-Side File Caching for MPI-IO

In our earlier work [13], we adopted a concept similar to cooperative caching and implemented a client-side file caching system as a library running entirely at the user space. Designed for MPI applications, this caching system uses the MPI communicator supplied to the file open call to identify the scope of processes that will later cooperate with each other to perform file caching. Previously, the caching system was implemented as a stand-alone library. Since then, we have incorporated it into the ADIO layer of ROMIO. The ADIO layer is where ROMIO interfaces with underlying file systems [22]. Here, we briefly describe the design and recent implementation updates.

3.1. I/O Thread

In order to have MPI processes cooperate with each other, the caching system needs a transparent mechanism in each process that can run independently and concurrently with the program main program. We choose an I/O thread approach. A POSIX thread is created in each MPI process when opening the first file and destroyed when closing the last file. Multi-threading enables cooperative caching without interrupting the main thread. This feature is particularly important for MPI applications that make independent I/O calls, since independent I/O requires no process synchronization. The I/O thread handles both local and remote requests to data cached in the local memory, and collaborates with remote threads for coherence control. Figure 1 illustrates the I/O thread's task flow inside an MPI process. The I/O thread communicates with the main thread through a mutex protected shared variable and uses `MPI_Iprobe()` to detect remote requests.

To deal with the fact that an MPI process may open more than one file, MPI-IO caching is added support to handle multiple files opened with different MPI communicators. The communicator of a newly opened file is added into a

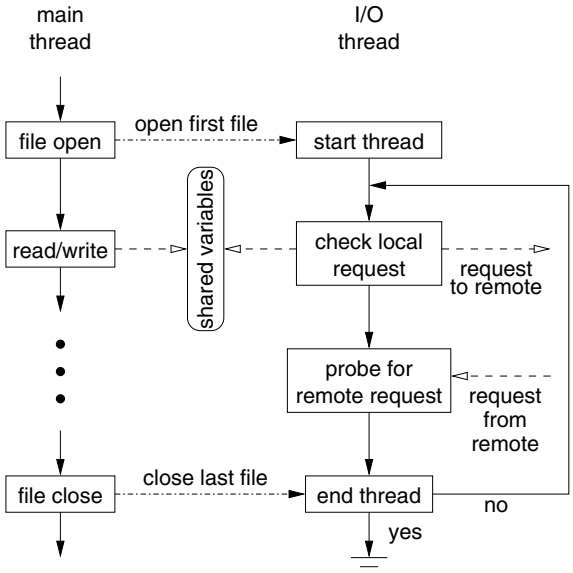

Figure 1. Design of MPI-IO caching from a single MPI process' view.

linked list in each process. In order to handle requests for all opened files, the I/O thread runs an infinite loop of calling `MPI_Iprobe()`, each with `MPI_ANY_SOURCE` as the source process and a communicator alternatively from the linked list. Blocking MPI communication functions, such as `MPI_Wait()` or `MPI_Probe()`, cannot be used here, because every MPI communication function must be bound with a communicator and the I/O thread must be active all the time to respond requests for any of the opened files. The blocking function called with a communicator prevents the I/O thread from detecting remote requests with a different communicator. In fact, our implementation uses only asynchronous MPI communication calls, so that waiting for one request to complete will not block the operations for processing another request.

3.2. Cache Metadata Management

In our implementation, a file is logically divided into equal-sized pages. The cache granularity is set to a page size whose default is the file system block size and changeable by an MPI hint. A page size aligned with the file block size is recommended, since it reduces the possibility of false sharing in the file system. Cache metadata, describing the cache status of these pages, are statically distributed in a round-robin fashion among the MPI processes that open the shared file collectively. Finding the process rank owning the metadata of a page requires only a modulus operation. This approach also avoids centralized metadata management. Previously, memory space for metadata

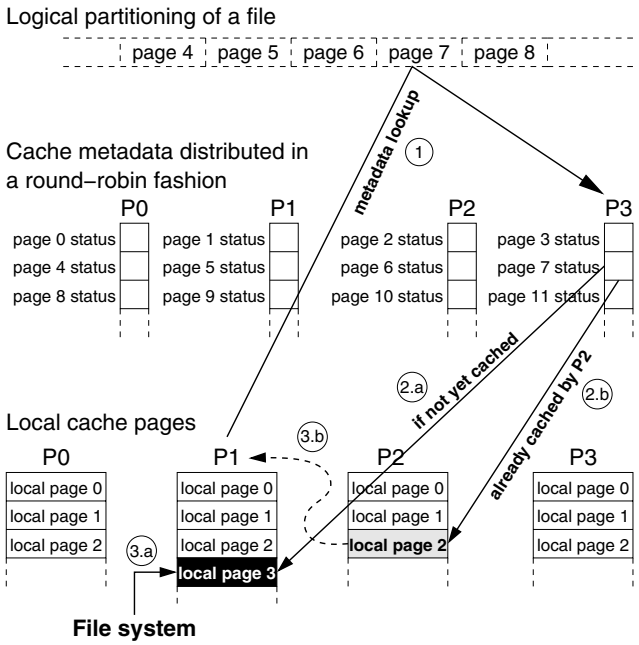

Figure 2. Example of the I/O flow in MPI-IO caching where MPI process P_1 reads data from logical file page 7.

is pre-allocated with a constant size. We have since changed it to allow increase of metadata space as the file size grows.

Cache metadata includes the page owner, MPI rank of the current location, locking mode, and the page's recent access history. A page's access history is used for cache eviction and page migration. We have added a new page migration mechanism to relocate a page if it is referred by the same remote process consecutively twice. The I/O flow of a read operation is illustrated in Figure 2 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.

3.3. Locking Protocol

To ensure cache metadata integrity, i.e. atomic access to metadata, a distributed locking mechanism is implemented. We let each MPI process be a lock manager for the assigned metadata and choose the file page size as the lock granularity to simplify the implementation. Locks only apply to metadata, in which locks must be granted to the requesting process before it can read/write the metadata. Once the metadata are locked, the MPI process is free to access the cache pages and the file range corresponding to the pages. This locking mechanism also provides a foundation for implementing MPI-IO atomicity. Most modern parallel file systems, such as GPFS and Lustre, adhere to the POSIX standard. Atomicity, as required by POSIX, defines the I/O consistency on a shared file as all bytes written by a single write call that are either completely visible or invisible to a following read call [10, 9]. Similar to POSIX, MPI-IO atomicity is required for each individual MPI read/write call. However, unlike the POSIX read/write functions that each can only access to a contiguous file region, a single MPI read/write can simultaneously access to multiple non-contiguous regions. To abide the MPI-IO semantics, scalable MPI-IO implementations must add an atomicity control layer to coordinate the non-contiguous reads/writes to guarantee the atomic result [20, 12]. As one can imagine, these implementations rely on the atomicity provided by the individual POSIX read/write.

In our MPI-IO caching where cache pages consecutive in file space can be stored at different MPI processes, I/O atomicity is guaranteed for a single POSIX read/write whose request spans more than two pages. To achieve this goal, we enforce the page locks for all read/write calls. In other words, a read/write call contain the operations in the order of getting locks, accessing cache pages, and releasing locks. Since all locks must be granted prior to accessing to the cache pages, dead locks may occur when more than two processes are concurrently requesting locks for two pages. To avoid dead locks, we employ the two-phase locking strategy proposed in [1]. Under this strategy, lock requests are issued in a strictly increasing order of page IDs and a page lock must be obtained before the next lock request is issued.

We have added two new implementations to improve locking. The first is use a pipelined locking strategy. In the earlier work, all page locks from a read/write call must be obtained prior to accessing the cache pages. The pipelined locking allows the access to the already-locked cache pages to proceed while waiting for other lock requests to be granted. However, to guarantee I/O atomicity, locks must be released altogether at the end of a read/write call. The second new implementation is to separate locks into sharable read locks and exclusive write locks. A counter is used to record the number of MPI processes sharing a read lock. The counter increases when the read lock is shared by a new process and decreases when released by a process. A write lock request will stop the counter from increasing and put itself and the subsequent read/write lock requests into a queue. Once the counter reaches zero, the write lock request is retrieved from the queue and granted exclusively.

3.4. Cache Page Management

To simplify coherence control, we allow at most a single copy of file data to be cached among all MPI processes. In our earlier implementation, a chunk of memory space is pre-allocated for file caching at the time the I/O thread is created. We have modified it to adopt a dynamic management method to adjust memory usage for caching based on available memory space. The caching policy is described as follows. 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. An upper bound, by default 64 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, i.e. under memory pressure, page eviction is activated. Eviction is solely based on the local references and a least-recent-used policy. If the requested file pages have not yet 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 has been cached locally, a memory copy can simply satisfy the request. If the page is cached at a remote process, the request is forwarded to the page owner.

When closing a file, all dirty cache pages are flushed to the file system. A high water mark is added for each cache page to indicating the range of dirty data, so that flushing needs not always be an entire page. Because contiguous logical file pages can potentially spread across all MPI processes, a new two-phase flushing function is devised during file close to mimic the two-phase I/O by shuffling 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. Performance Evaluation

Our implementation for MPI-IO caching was 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 with a shared 3 GB 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. To store the output files, we created a directory with the configuration of 64 KB stripe size and 8 I/O servers. All files saved in this directory shared the same striping configuration. Mercury is a 887-node IBM Linux cluster where each node contains two Intel 1.3/1.5 GHz Itanium II processors with a shared 4 GB 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. Note that the IBM's MPI is not available on Mercury and hence we did not evaluate the performance of GPFS's data shipping. Regarding thread-safety, our MPI-IO caching was implemented in the ROMIO layer of the MPICH version 2-1.0.3, the thread-safe and latest version of MPICH2, at the time our experiments were performed. However, thread-safety is only supported for the default sock channel of MPICH2. The inter-process communication in our experiments used Gigabit Ethernet, which is relatively slower than the Myrinet on the same machines. To evaluate MPI-IO caching, we used BTIO and FLASH I/O benchmarks.

Developed by NASA Advanced Supercomputing Division, the parallel benchmark suite NPB-MPI version 2.4 I/O is formerly known as the BTIO benchmark [25]. BTIO presents a block-tridiagonal partitioning pattern on a three-dimensional array across a square number of compute nodes. Each processor 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 four I/O methods: MPI collective I/O, MPI independent I/O, Fortran I/O, and separate-file I/O. There are 40 consecutive collective MPI writes and each appends an entire array to the previous write in a shared file. The writes are followed by 40 collective reads to verify the newly written data. We evaluated two I/O sizes: classes B and C with array dimensions of $102 \times 102 \times 102$ and $162 \times 162 \times 162$, respectively.

The FLASH I/O benchmark suite [27] is the I/O kernel of 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 [5]. The computational domain is divided into blocks which are distributed across the 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 this work, we used two block sizes of $16 \times 16 \times 16$ and $32 \times 32 \times 32$. There are 24 variables per array element, and about 80 blocks on each MPI process. A variation of block numbers per MPI process is used to generate a slightly unbalanced I/O load. Since the number of blocks is fixed in each process, as we increase the number of MPI processes, the aggregated I/O amount linearly increases as well. FLASH I/O produces a checkpoint file and

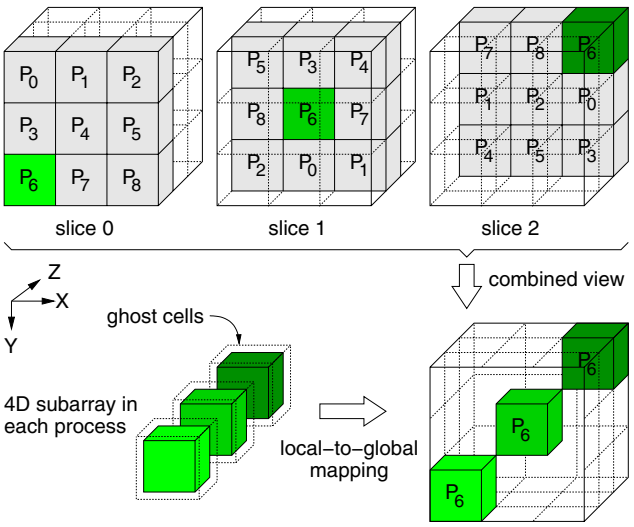

Figure 3. BTIO data partitioning pattern. The 4D subarray in each process is mapped to the global array in a block-tridiagonal fashion. This example uses 9 processes and highlights the mapping for process P_6.

two visualization files containing centered and corner data. The I/O uses HDF5, which allows data to be stored along with its metadata in the same files. HDF5 is built on top of MPI-IO [6]. To eliminate the overhead of memory 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 passing it into HDF5 routines. 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. Inside ROMIO, this non-interleaved access pattern actually triggers the subroutines used by independent I/O, even if MPI collective writes are called.

Table 1 shows the I/O amounts for both BTIO and FLASH I/O used in our experiments. We believe that the performance evaluation for a caching system is different from measuring the maximum data rate for a file system. Typical file system benchmarks avoid caching effect by using an I/O amount larger than the aggregated memory size of either clients or servers. File caching can only be beneficial when there is sufficient unused memory space for the caching system to operate in. Therefore, we use the medium array sizes for the benchmarks in our experiments such that the I/O amount does not overwhelm the memory size of compute nodes. In this paper, we report the aggregate I/O bandwidth, since all inter-process communications in our implementations use MPI asynchronous functions and it is very hard to separate the costs for computation, communication, and file I/O. The I/O bandwidth numbers were obtained by dividing the aggregate read/write amount by the time measured from the beginning of `MPI_File_open()` until after `MPI_File_close()`. Note that although no explicit `MPI_File_sync()` call is made in both benchmarks, closing files will flush all dirty cache data.

Table 1. The I/O amount (in GB) of BTIO and FLASH I/O benchmarks.

	BTIO		FLASH I/O	
no. nodes	array dim.		array dim.	
	102^3	162^3	16^3	32^3
16	3.24	12.97	1.15	9.13
25	3.24	12.97	-	-
32	-	-	2.30	18.26
36	3.24	12.97	-	-
49	3.24	12.97	-	-
64	3.24	12.97	4.60	36.53

4.1. Compare with File System Caching

We first compare the benchmarks with and without MPI-IO caching. We refer *native* cases as the results of running the benchmarks without MPI-IO caching. Figure 4 shows the aggregate I/O bandwidths using up to 64 compute nodes. The cases of "caching + o_direct" will be described in the next section. For BTIO, MPI-IO caching improves I/O bandwidth about 10 to 20 times on Lustre and 2 to 5 times on GPFS. The improvement on GPFS is moderate and reasonable for the caching effect expected in BTIO's read-after-write pattern. We believe the improvement is hardly the effect of reading locally cached data at the read phase, but the reduction of file system's lock contention. Both Lustre and GPFS are POSIX compliant file systems and therefore respect POSIX I/O atomicity semantics. To guarantee atomicity, parallel file systems often enforce file locking in each read/write call to gain exclusive access to the requesting 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. However, it is known that lock contention due to enforcing atomicity can significantly degrade parallel I/O performance [20, 12].

Lustre employs a server-based distributed file locking protocol where each I/O server is responsible for managing locks for the file stripes it controls. The lock granularity on Lustre is the system page size, 4 KB on Tungsten. GPFS uses a token-based distributed file locking protocol, as described in Section 2. The lock granularity on GPFS is the disk sector size, 512 bytes on Mercury. If two writes are not

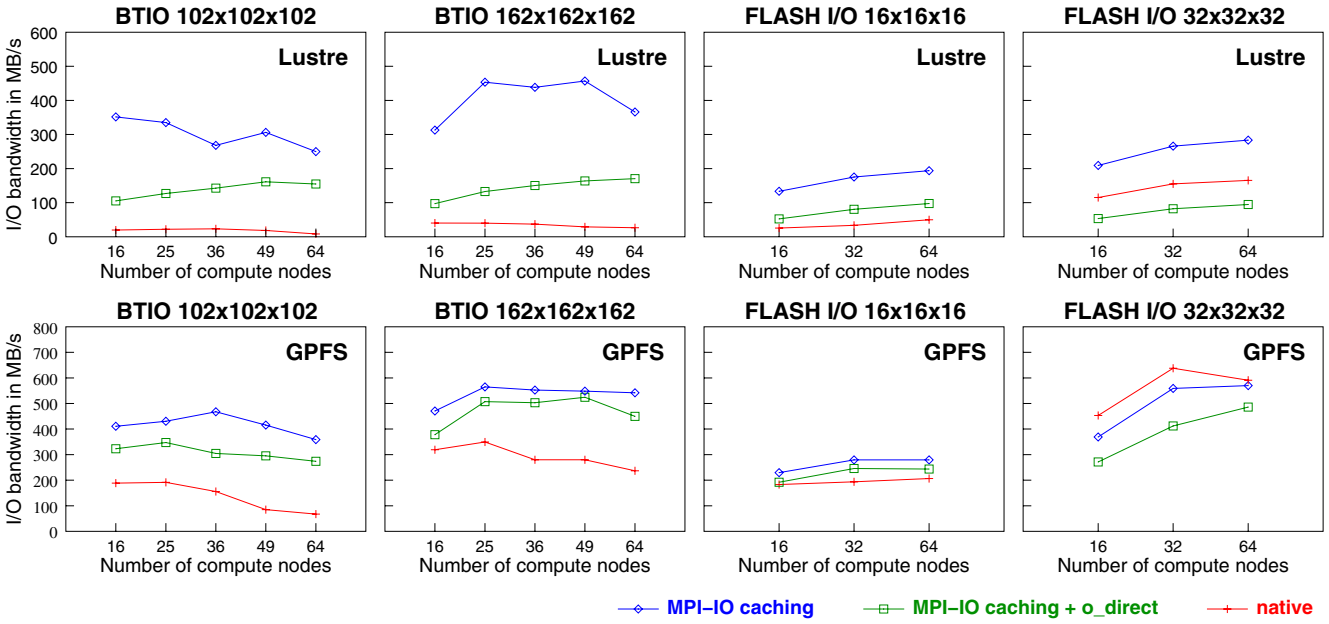

Figure 4. I/O bandwidth results of BTIO and FLASH I/O benchmarks on Lustre and GPFS.

overlapping in byte range but unaligned on the lock granularity boundaries, then they can still cause contention. Both BTIO and FLASH I/O used in our experiments only generate read/write calls with offsets and lengths that both are neither aligned to the boundaries of file stripe size nor the system page size. Hence, lock contention occurs in every two processes with consecutive MPI ranks, which results in a serious I/O serialization. Comparing the performance between Lustre and GPFS, the token-based protocol seems to have a better I/O parallelism. This effect can also be observed from the FLASH I/O results. On the other hand, MPI-IO caching can avoid such lock contentions if the logical file page size (cache page size) is chosen a multiple of file stripe size. In this case, all system I/O operations in MPI-IO caching are within the boundaries of file stripe size. In our experiments, we used 1024 KB cache page size, a multiple of the file stripe size on both Lustre and GPFS.

For FLASH I/O, MPI-IO caching's improvement is not as significant as BTIO. In the $32 \times 32 \times 32$ array size case on GPFS, the native MPI-IO is even better than the MPI-IO caching. As mentioned earlier in Section 1, the advantage of file caching exists in two scenarios: when repeated access is presented, and when write behind results in better network utilization. Note that MPI-IO caching bears several overheads, including memory copying between I/O buffers and cache pages, distributed lock management, and communication for remote accessing cache pages. Since FLASH I/O performs write-only operations and shows no repeating access pattern, write behind becomes the sole factor that can contribute a better performance for MPI-IO caching. When the write amount becomes larger, as the array size increased from $16 \times 16 \times 16$ to $32 \times 32 \times 32$, file caching loses the ground for the write-behind advantage.

4.2. Bypass File System Caching

On both Lustre and GPFS, system-level client-side caching is enabled by default. Therefore, the performance numbers of MPI-IO caching shown in Figure 4 actually were the results of two levels of caching. It would be interesting to see how MPI-IO caching performs when system-level caching is disabled. We found that direct I/O is a portable approach to bypass system caching. Enabled by adding the O_DIRECT flag at the time a file is opened, direct I/O allows read/write calls to be carried out directly to/from user I/O buffers. However, it requires a read/write call's file offset, request length, and user buffer address be multiples of the system's logical block size. On Lustre, it is the system page size, 4 KB, and on GPFS, it is the disk sector size, 512 bytes. If the above requirements are not met, then Lustre will return an EINVAL error and GPFS will automatically turn into regular, non-direct I/O. It is advised that designed to minimize cache effects, direct I/O is only useful if applications do their own caching. While in direct I/O mode, all I/O calls are synchronous, i.e., at the completion of a read/write call, data is guaranteed to have been transferred.

Since MPI-IO caching only generates reads/write requests aligned with the cache page size, direct I/O seems fit to MPI-IO caching naturally if the cache page size is a mul-

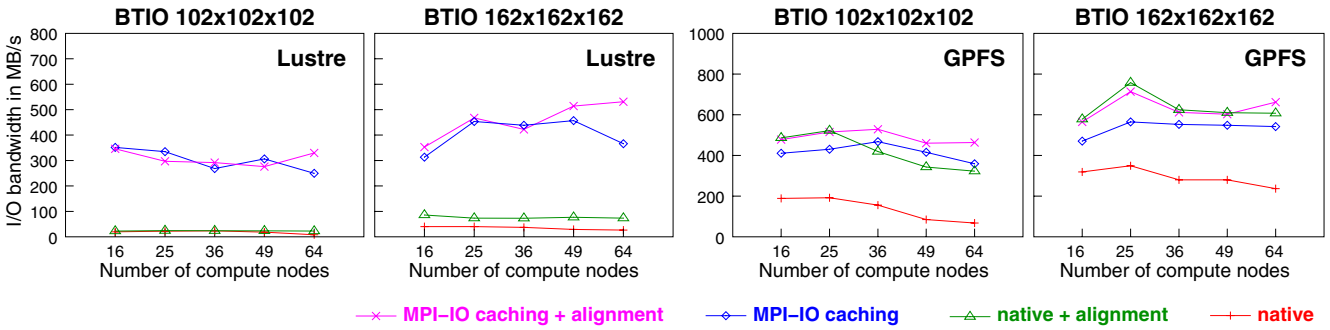

Figure 5. BTIO bandwidth with file domain alignment.

tiple of system block size. The experimental results of MPI-IO caching with direct I/O are added in Figure 4. Note that direct I/O for the native cases, i.e. MPI-IO through POSIX reads/writes, are not available because the block alignment requirements on the buffer address, file offset, and length cannot be fulfilled by BTIO and FLASH I/O natively. The only way to use direct I/O in the native cases is to reallocate the user buffers whenever they are not aligned with the block size. In addition, if the file offset is not aligned, a read-modify-write operation at the page boundaries cannot be avoided. One can expect such cost of memory reallocation and read-modify-write operations to easily overwhelm the advantages of direct I/O.

As seen in Figure 4, the results of MPI-IO caching with direct I/O do not further improve the performance. However, they are still better than the native cases, except for the case of the FLASH I/O with $32 \times 32 \times 32$ array size. In fact, direct I/O enables a few welcome features for MPI-IO caching. First of all, the memory copy operations between user buffers and system cache are eliminated. On GPFS, even internal byte-range locking for cache coherence control is automatically disabled when doing direct I/O. Furthermore, no prefetching is done by GPFS, either. On Lustre, we explicitly disabled file locking via a call to `ioctl()`, which removes lock contention protocols. Note that it is only safe to disable file locking on Lustre when doing direct I/O. Nevertheless, these advantages cannot overcome the fact that all reads/writes are synchronous. On both GPFS and Lustre, such synchronization goes further to the disks on the I/O servers, not just servers' memory, before a write call returns. On the other hand, without direct I/O, data can be cached at the I/O servers so that a write call can return immediately after the servers receive the data. If servers' aggregated memory space is large enough, write data can entirely reside in servers' memory without touching the disks, even after closing the file. In our experiments, the disk synchronization cost clearly overwhelms the benefits of the eliminating lock contention, buffer copying, and prefetching.

4.3. File Domain Alignment

As discovered previously in section 4.1, lock contention occurs when I/O requests are not aligned at the boundaries of the file system's lock granularity. Hence, we studied the effect of alignment to both I/O benchmarks. ROMIO adopts the two-phase I/O strategy proposed in [4] in its implementation for MPI collective I/O functions. Two-phase I/O consists of an I/O phase and a communication phase. At first, an aggregate access region is formed as a contiguous range that covers all I/O requests from all MPI processes. *File domains* are then defined and calculated by evenly dividing the aggregate access region by the number of MPI aggregators. The division is done at the byte range granularity. Designated aggregators can be a subset of or all the MPI processes that opened the shared file collectively. Note that the term file domain is only valid in MPI collective I/O operations (two-phase I/O, to be precise), which is a contiguous region to which the owning process has exclusive access. In the I/O phase, each MPI aggregator makes read/write calls 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 calculation of file domains currently used in ROMIO tends to achieve a balanced I/O workload among the aggregators. However, balanced workload may not always result in aligned I/O requests to reduce lock contention.

We modified ROMIO's two-phase I/O implementation to allow the division of file domains to align with the file stripe boundaries. This implementation is located at the beginning of two-phase I/O and is completely independent from MPI-IO caching. Prior to calculating the alignment, the file's stripe size must be known. Usually, the function call to query stripe size is file system dependent. On Lustre, a file's stripe size can be obtained via a call to `llapi_file_get_stripe()` from the Lustre's API library. On GPFS, function `gpfs_fstat()` tells the system file block size, which is also the stripe size in most of the GPFS configurations. Similar file domain alignment has

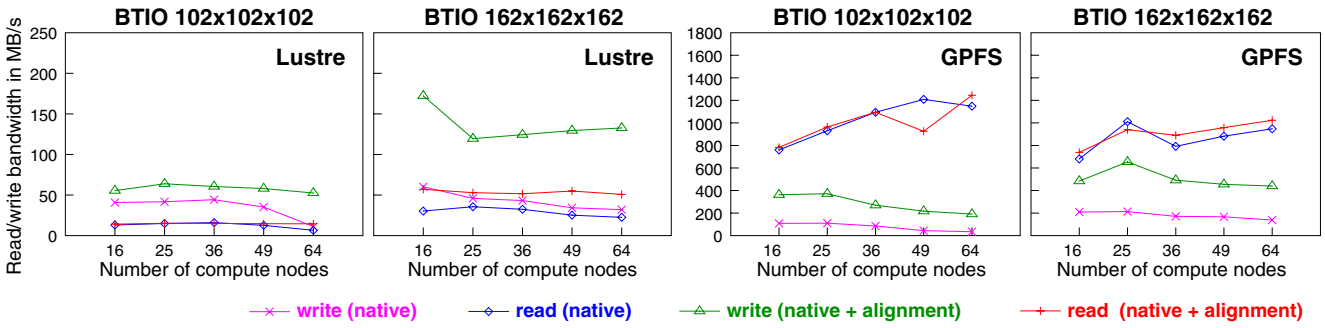

Figure 6. BTIO read/write bandwidths with file domain alignment for the native case.

also been tested on the IBM BlueGene/L GPFS experiments [26] and demonstrated an improved performance. The performance results are given in Figure 5. We aligned the file domains with 64-KB boundaries on Lustre and 512-KB on GPFS (both are the system stripe sizes). No FLASH I/O results are presented because its non-interleaved write pattern will only trigger ROMIO independent I/O subroutines internally, even if MPI collective writes are called in the applications. Since file domains are only defined in ROMIO's collective I/O subroutines, file domain alignment is not applicable to FLASH I/O.

When comparing aligned and unaligned MPI-IO caching, we observe fluctuant but similar bandwidths on Lustre and a noticeable improvement for aligned MPI-IO caching on GPFS. File domain alignment benefits MPI-IO caching further due to the removal of conflict locks internal to the MPI-IO caching's locking protocol. These conflict locks can only occur at the boundaries of cache pages. Nevertheless, the alignment effects on the native cases are especially interesting. When comparing aligned and unaligned native cases, the alignment shows a slight improvement on Lustre and a noticeable bandwidth escalation on GPFS. The bandwidths of aligned native cases even become comparable to the MPI-IO caching on GPFS. This phenomenon attributes to the complete removal of write lock contention that enables GPFS to cache BTIO's data at the clients' memory with minimal coherence control overhead.

At first glance, the alignment's improvement for the native cases on Lustre seems not as significant as on GPFS. However, when looking at the I/O bandwidth numbers for BTIO class C, we observed 2 to 3 times improvement ratios both on Lustre and GPFS. This implies that the alignment has a similar performance impact to the native cases on both file systems. These ratios may not be obvious in Figure 5 because of Lustre's relatively lower bandwidth. One of the reasons causing the poor performance in general on Lustre is its system-level read-ahead operation. Read ahead is activated on Lustre by default. From a single MPI process's view, each read operation in BTIO results in an access to a file space that is not contiguous from the previous read, which renders the pre-fetched data completely useless. As shown in Figure 6, the native case's read and write bandwidths were measured separately and much lower read bandwidths were observed than the write in all BTIO cases on Lustre. Performance degradation due to read ahead on Lustre has also been reported in [7]. Nevertheless, more investigation is still needed for explaining Lustre's poor performance for the parallel I/O patterns used in BTIO and FLASH I/O.

5. Conclusions

There is no easy way to justify the parallel I/O performance of a caching system, given non-obvious influences such as file locking for I/O atomicity and file domain alignment to the system stripe size, as discussed in this paper. We choose BTIO and FLASH I/O to evaluate MPI-IO caching because their parallel I/O patterns are more realistic than many artificial benchmarks and applications in the real world may not always generate aligned I/O requests. In fact, the nature of their unaligned patterns helped us to discover hidden factors in parallel file system design that hamper I/O performance. Enforcing POSIX semantics such as I/O atomicity has become an obstacle for parallel file systems providing an I/O rate close to the maximum network bandwidth. Many large-scale parallel applications primarily use write-only and non-overlapping patterns, which do not require strict semantics like POSIX atomicity. In our MPI-IO caching design, file locking is also used to enforce atomicity. We plan to design and add a mechanism as a user option to relax atomicity. We conservatively chose to keep at most a single copy of file data within the MPI processes' memory. Although it simplifies coherence control, this restriction can increase the communication cost for remote cache page access. 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 rebalancing and data prefetching.

6. Acknowledgments

This work was supported in part by Sandia National Laboratories and DOE under contract number 28264, DOE's SciDAC program (Scientific Data Management Center), award number DE-FC02-01ER25485, NSF's NGS program under grant CNS-0406341, NSF/DARPA ST-HEC program under grant CCF-0444405, NSF HECURA CCF-0621443 and the National Center for Supercomputing Applications under DDM050006 and CCR060023, and utilized the Dell Xeon Cluster and IBM IA-64 Linux Cluster.

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, Aug. 1996.

[3] M. Dahlin, R. Wang, and T. A. adn D. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *the First Symposium on Operating System Design and Implementation*, Nov. 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, Apr. 1993.

[5] 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.

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

[7] R. Hedges, B. Loewe, T. McLarty, and C. Morrone. Parallel File System Testing for the Lunatic Fringe: the care and feeding of restless I/O Power Users. In *the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies*, Apr. 2005.

[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 Std. 1003.1-2001. *System Interfaces*, 2001.

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

[11] 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.

[12] W. Liao, A. Choudhary, K. Coloma, G. Thiruvathukal, W. Lee, E. Russell, and N. Pundit. Scalable Implementations of MPI Atomicity for Concurrent Overlapping I/O. In *the International Conference on Parallel Processing*, Oct. 2003.

[13] 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.

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

[15] 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)*, Apr. 2003.

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

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

[18] 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*, Nov. 2001.

[19] 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*, Aug. 2000.

[20] 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.

[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, Jan. 2002.

[22] 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*, Oct. 1996.

[23] 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, Oct. 1997.

[24] 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.

[25] 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, Jan. 2003.

[26] H. Yu, R. Sahoo, C. Howson, G. Almasi, J. Castanos, M. Gupta, J. Moreira, J. Parker, T. Engelsiepen, R. Ross, R. Thakur, R. Latham, and W. D. Gropp. High Performance File I/O for the BlueGene/L Supercomputer. In *the 12th International Symposium on High-Performance Computer Architecture (HPCA-12)*, Feb. 2006.

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