Cooperative Client-Side File Caching for MPI Applications

Wei-keng Liao, Kenin Coloma, Alok Choudhary and Lee Ward

International Journal of High Performance Computing Applications 2007 21: 144
DOI: 10.1177/1094342007077857

COOPERATIVE CLIENT-SIDE FILE CACHING FOR MPI APPLICATIONS

Wei-keng Liao[1]
Kenin Coloma[1]
Alok Choudhary[1]
Lee Ward[2]

Abstract

Client-side file caching is one of many I/O strategies adopted by today's parallel file systems that were initially designed for distributed systems. Most of these implementations treat each client independently because clients' computations are seldom related to each other in a distributed environment. However, it is misguided to apply the same assumption directly to high-performance computers where many parallel I/O operations come from a group of processes working within the same parallel application. Thus, file caching could perform more effectively if the scope of processes sharing the same file is known. In this paper, we propose a client-side file caching system for MPI applications that perform parallel I/O operations on shared files. In our design, an I/O thread is created and runs concurrently with the main thread in each MPI process. The MPI processes that collectively open a shared file use the I/O threads to cooperate with each other to handle file requests, cache page access, and coherence control. By bringing the caching subsystem closer to the applications as a user space library, it can be incorporated into an MPI I/O implementation to increase its portability. Performance evaluations using three I/O benchmarks demonstrate a significant improvement over traditional methods that use either byte-range file locking or rely on coherent I/O provided by the file system.

Key words: client-side file caching, parallel I/O, MPI I/O, cache coherence, I/O thread

The International Journal of High Performance Computing Applications,
Volume 21, No. 2, Summer 2007, pp. 144–154
DOI: 10.1177/1094342007077857

1 Introduction

In today's high-performance computers, file systems are often configured in a client–server model. There are usually far fewer I/O servers than application nodes since system design targets primarily computationally intensive applications. In such a scenario, potential communication bottlenecks can form at the server nodes when large groups of compute nodes make file requests simultaneously. Therefore, reducing the amount of data transfer between clients and servers becomes an important issue for parallel file system design. As demonstrated by distributed file systems, client-side file caching is often considered a technique that allows scaling and is adopted in many modern parallel file systems.

File caching places a replica of repeatedly accessed data in the memory of the requesting processors such that successive requests to the same data can be carried out locally without going to the file servers. However, storing multiple copies of the same data at different clients introduces the cache coherence problem (Tanenbaum and van Steen 2002). Existing system-level solutions often involve the bookkeeping of cache status data at the I/O servers and require that I/O requests first consult the servers for safety to proceed. Such coherence control mechanisms require file locking in each read/write request to ensure atomic access to the cache data. Since file locking is usually implemented in a centralized manner, it can easily limit the degree of I/O parallelism for concurrent file operations.

We propose a cooperative client-side file caching scheme for MPI applications that access shared files in parallel. The motivation comes from the inappropriate assumption within traditional client-side caching designs that considers each I/O request independently without correlation between the requesting clients. While such an assumption may be suitable for distributed file systems, it can aggravate the cache coherence problem for parallel applications that work on the same data structures and perform concurrent I/O to shared files. Cooperative caching instead coordinates the MPI processes that open the same file collectively to perform file caching and coherence control. The design consists of the following five components: 1) cache metadata describing the caching status of the file; 2) a global cache pool comprising local memory from all MPI processes; 3) I/O threads running concurrently with the main program in each MPI process; 4) a distributed locking

[1]ELECTRICAL ENGINEERING AND COMPUTER SCIENCE DEPARTMENT, NORTHWESTERN UNIVERSITY, EVANSTON, IL 60208, USA (WKLIAO@ECE.NORTHWESTERN.EDU)

[2]SCALABLE COMPUTING SYSTEMS DEPARTMENT, SANDIA NATIONAL LABORATORIES, ALBUQUERQUE, NM 87185, USA

mechanism; and 5) caching/eviction policies. In our current implementation, the caching system is built as a user-level library that can be linked by MPI applications. We plan to incorporate it into an MPI I/O implementation in the future. Experimental results presented in this paper were obtained from the IBM SP at San Diego supercomputing center using its GPFS file system. Three sets of I/O benchmarks were used: a sliding-window I/O pattern, BTIO, and FLASH I/O. Compared with the traditional approaches that use either byte-range file locking to enforce the cache coherence or simply native UNIX read/write calls, cooperative caching proves to be a significant performance enhancement.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 describes the design and implementation of the cooperative caching. The performance results are presented in Section 4 and the paper is concluded in Section 5.

2 Related Work

Many parallel file systems such as GPFS (Schmuck and Haskin 2002), Lustre (Cluster File Systems 2003), and Panasas (Nagle, Serenyi, and Matthews 2004) support client-side caching. However, targeting different I/O patterns, parallel file systems adopt different coherence control strategies. GPFS employs a distributed lock protocol to enforce coherent client-side cache data. Lock tokens must be granted before any I/O operation can be performed on the cache data (Prost et al. 2000; 2001). To avoid centralized lock management, a token holder becomes a local lock manager responsible for granting locks for any further requests to its particular byte ranges. IBM's MPI I/O implementation for GPFS adopts a strategy called data shipping that binds each GPFS file block to a unique I/O agent responsible for all the accesses to this block. The file block assignment is done in a round-robin striping scheme. I/O operations must go through the I/O agents which ship the requested data to the appropriate processes. I/O agents are threads residing in each MPI process and are responsible for combining I/O requests in collective operations. In the background, I/O threads also assist in advanced caching techniques such as read ahead and write behind.

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 owns. If a client requests a lock held by another client, a message is sent to the lock holder asking for release of the lock. Before a lock can be released, dirty cached data must be flushed to the servers. The Panasas file system also employs metadata servers to maintain client-side cache coherence through using callbacks. A client's read or write request is first registered with the servers with a callback. If conflicting cache data access occurs, all clients that have callbacks are contacted and the appropriate write-back or cache invalidation is performed. In this scheme, data is transferred between the servers each time coherence control is applied. Some parallel file systems do not even support client-side file caching at all, for instance PVFS (Carns et al. 2000).

Cooperative caching is proposed by Dahlin et al. (1994) to provide coherence by coordinating clients' caches and allowing requests not satisfied by a client's local cache to be satisfied by the cache of another client. Systems that use cooperative caching are PGMS (Voelker et al. 1998), PPFS (Huber et al. 1995), and PACA (Corts, Girona, and Labarta 1996). The Clusterfile (Isaila et al. 2004) parallel file system integrates cooperative caching into MPI collective I/O operations by using cache managers to manage a global cache consisting of memory buffers on both clients and servers. However, cooperative caching in general requires changes in the file system at both client and server. In contrast, the caching scheme proposed in this work is implemented in user space and requires no change to the underlying file system.

2.1 MPI I/O

The Message Passing Interface (MPI) standard (MPI Forum 1995) defines an application programming interface for developing parallel programs that explicitly use message passing to perform inter-process communication. MPI version 2 (MPI Forum 1997) extends the interface to include, among other things, file I/O operations. MPI I/O 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. In general, there are two types of MPI I/O data access operations, collective I/O and independent (non-collective) I/O. Collective operations require all the processes that opened the file to participate. Thanks to the explicit synchronization, many collective I/O implementations take this opportunity to exchange access information among all the processes to generate a better overall I/O strategy. An example of this is the two-phase I/O technique proposed by del Rosario, Brodawekar, and Choudhary (1993). In contrast, independent I/O does not require synchronization, making any cooperative optimizations very difficult.

Active buffering proposed by Ma et al. (2003) is considered an optimization for MPI collective write operations. It accumulates write data in a local buffer and uses

an I/O thread to perform write requests in the background. I/O threads can dynamically adjust the size of local buffers based on 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 write behind enhances performance, active buffering is applicable only if the I/O patterns consist only of collective writes. Because of the lack of consistency control, active buffering cannot handle the mixed read and write operations or mixed independent and collective I/O calls.

3 Design and Implementation

The idea behind our cooperative caching is moving the caching system that is traditionally embedded in the operating system up to the user space and letting application processes cooperate with each other to manage a coherence. One of the immediate benefits is relieving I/O servers of the cache coherence control burden. Once data reaches the clients, client processes put forth the effort to keep the cached data coherent. As a result, I/O servers are solely responsible for transferring data to and from the clients, and no longer need to enforce cache coherence. Designed for MPI applications, our caching system uses the MPI communicator supplied in the file open call to identify the scope of processes that will later cooperate with each other to perform file caching. We now describe the building blocks of the caching system: management for cache metadata and cache pages, the I/O thread mechanism, distributed locking layer, and caching policies.

3.1 Cache Metadata and Cache Pages

In our implementation, a file is logically divided into pages of equal size where each page represents an indivisible unit that can be cached in a process' local memory. Describing the cache status of these file pages, cache metadata is distributed in a round-robin fashion among the processes in the MPI communicator that opened the file. Thus, metadata for page *i* resides on the process with rank (*i* mod *nproc*), where *nproc* is the number of processes in the MPI communicator. Figure 1 depicts an example distribution of cache metadata and cache pages for a file opened by four MPI processes. Note that the location of cache metadata is fixed (cyclically assigned among processes) but file pages can be cached at any process. Cache metadata includes the page owner, MPI rank of the cur-

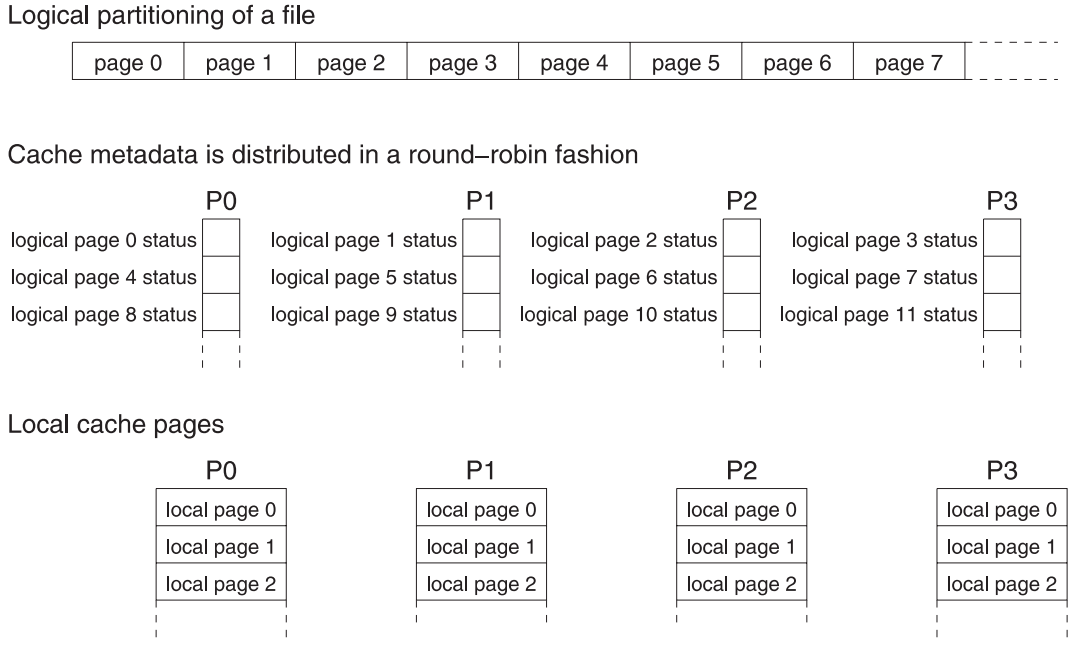

Fig. 1 An example of cache metadata distribution in the cooperative caching among four processes. A file is logically divided into equal-sized pages. The cache status of the logical pages is distributed in a round-robin fashion. The local cache pages are a collection of memory buffers used for caching. If cached, the contents of an entire logical file page are stored in a local file page. All local cache pages are of the same size as the logically partitioned file page.

rent location, locking mode (locked/unlocked), and the page's recent access history. For each page cached locally, we use a high water mark to indicate the dirty data in the page.

The cache granularity is a file page size. Our current implementation pre-allocates a fixed size of memory for each process to store the cache pages. These memory spaces from all the involved MPI processes can be viewed as a global cache pool. Accessing a cache page in the global pool can only be done through either local memory copy or explicit MPI communication. Note that the cache pool is shared by all the opened files, but cache metadata is created uniquely for each file. An I/O request must first check the status of each of its pages prior to accessing the pages or the file region corresponding to the pages.

3.2 I/O Thread

Since cache pages and metadata are distributed among processes, each process must be able to respond to remote requests to access data stored locally. Because collective I/O is inherently synchronous, remote queries can be fulfilled easily with inter-process communication. The fact that independent I/O is asynchronous makes it difficult for any one process to explicitly handle arbitrary remote requests. Hence, we choose an approach that creates an I/O thread to run concurrently with the main program thread. To improve the portability, our implementation uses the POSIX thread library (IEEE 1996). Figure 2 illustrates the I/O thread design within an MPI process. Details of the I/O thread design are described as follows.

- Each process can have multiple files opened, but only one thread is created. The I/O thread is created when the process opens the first file and destroyed when the last file is closed.
- Once the I/O thread is created, it enters an infinite loop to serve both local and remote I/O requests until it is signaled by the main thread for its termination.
- All I/O and communication operations related to caching are carried out by the I/O thread only. For blocking I/O operations, the main thread signals the I/O thread and waits for the I/O thread to complete the request. For non-blocking I/O, the main thread can continue its task, but must explicitly call a wait function to ensure the completion of the request. This design conforms to the MPI blocking and non-blocking I/O semantics.
- A shared conditional variable protected by a mutual exclusion lock is used to indicate if an I/O request has been issued by the main thread or if the I/O thread has completed the request. The communication between the two threads is done through a few shared variables depicting file access information such as the file handler, offset, memory buffer, etc.

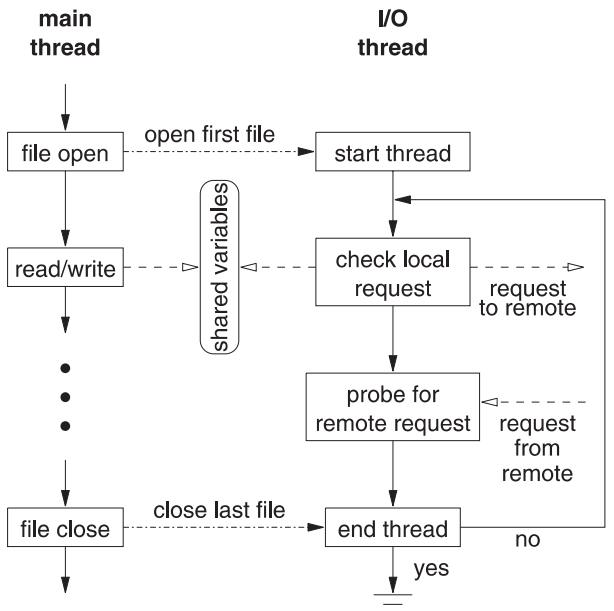

Fig. 2 I/O thread's interactions with the main thread and remote requests from a single MPI process' point of view.

- To serve remote requests, the I/O thread probes for incoming I/O requests from all processes in the MPI communicator group. Since each opened file is associated with a communicator, the probe will scan all the opened files.
- The types of local requests are: file open, close, read, write, flush, and thread termination.
- The remote I/O requests include cache page data transfers, and read/write, lock/unlock metadata.

3.3 Distributed Locking Layer

Since cache metadata may be modified at any time, a distributed locking mechanism is implemented to ensure access is atomic. Each MPI process acts as a lock manager for its assigned metadata with lock granularity matching the file page size to simplify the implementation. This design concept is similar to the distributed locking protocol used in existing parallel file systems, such as Lustre. Locks only apply to metadata, so locks must be granted to the requesting process before it can read/write the metadata. If the requested metadata is currently locked, the request will be added to a queue and the requesting process must wait for the lock to be granted. Once the metadata locks are granted, the MPI process is free to access the metadata, cache pages, and the file range corresponding to the pages.

Our implementation follows the POSIX semantics on I/O atomicity. POSIX atomicity requires that all bytes written by a single write call are either completely visible or completely invisible to any read call (IEEE 1996; 2001). Similar to POSIX, MPI I/O consistency 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. In an MPI I/O implementation, each of these non-contiguous regions must be carried out by a single read/write call, unless the underlying file systems support accessing multiple non-contiguous regions in a single call. Therefore, atomicity for a single contiguous access is important for an MPI I/O implementation to enforce the I/O consistency. In cooperative caching, cache pages in consecutive file space can be stored at different MPI processes, so I/O atomicity must be guaranteed for a single POSIX read or write whose request spans multiple pages. To achieve this end, locks are required for all read and write calls. In other words, a read or write calls consist of getting locks, accessing cache pages, and releasing locks. Since all locks must be granted prior to accessing the cache pages, deadlock may occur when more than two processes concurrently request locks to the same two pages. To avoid deadlock, we employ the two-phase locking strategy proposed by Bernstein, Hadzilacos, and Goodman (1987). Under this strategy, lock requests are issued in strict ascending order on page IDs and a page lock must be obtained before issuing the next lock request. For example, if a read or write call covers the file pages from i to j, where $i \le j$, the lock request for page k, $i \le k \le j$, will not be issued until the lock for page $(k-1)$ is granted.

3.4 Caching Policies

To simplify coherence control, at most one copy of file data can be cached. The operations for a read request are described in Figure 3(a). When a process makes an I/O call, its main thread first sets the access information in the shared variables and signals the I/O thread. Once signaled, the I/O thread uses the current file pointer position and the request length to identify the file page range. For each file page, the I/O thread sends a lock request to the process that holds the metadata. The lock to the metadata must be granted before any read/write can proceed on the file range corresponding to the page. The locks are only applicable to the metadata rather than the cache pages. If the requested pages have not been cached by any process, the requesting process will cache them locally by reading them from the file system. Otherwise, the requests are forwarded to the owner(s) of the cache pages.

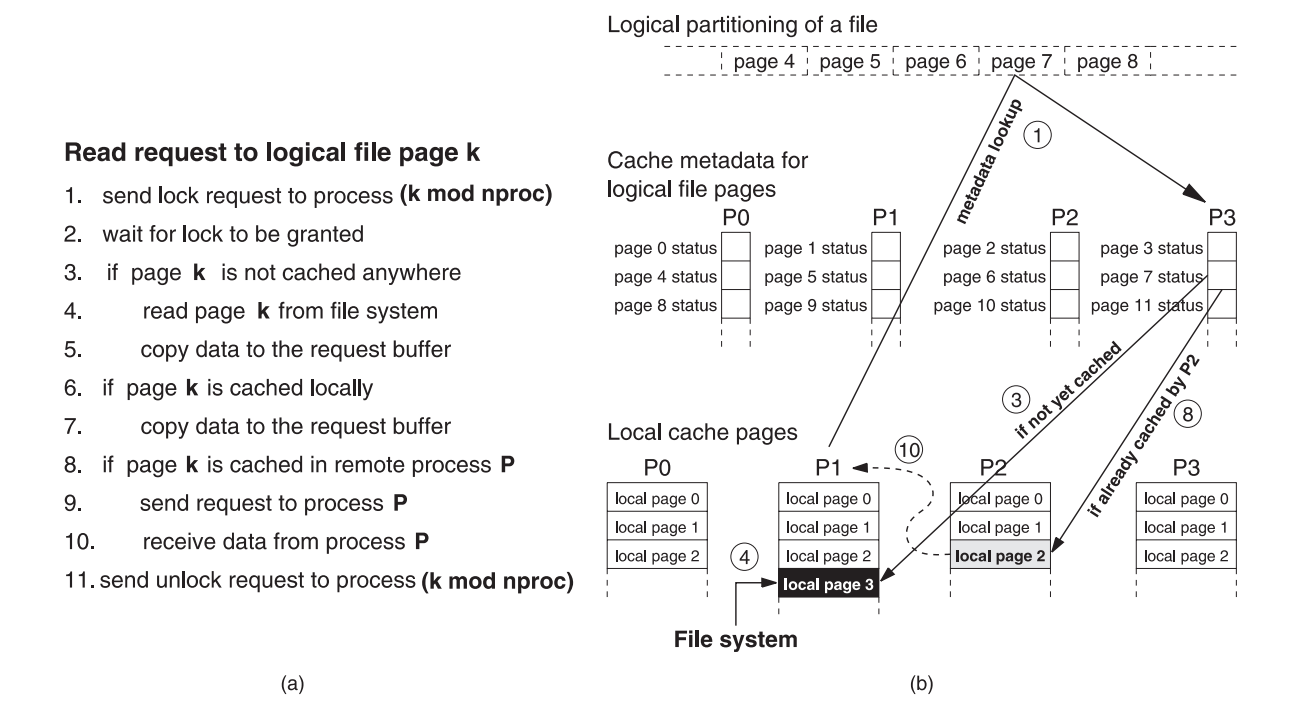

Fig. 3 (a) The operations for a read request to file page *k*. (b) Example of the I/O flow where MPI process P_1 reads data from logical file page 7.

The I/O flow of a read operation is illustrated in Figure 3(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 (3) and (4) in Figure 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 (10), assuming file page 7 is cached in local page 2, P_2 sends the requested data to P_1. In our implementation, data transfer between two I/O threads on different MPI processes is through MPI asynchronous communication. We have also been investigating an alternative design based on the MPI remote memory access functions. For this work, the reader is referred to Coloma et al. (2005).

In our implementation, cache data is flushed under any of three conditions: under memory pressure, at file close, and the explicit file flush function. Page eviction is initiated when the pre-allocated memory space is full. The eviction policy is solely based on local references and a least-recent-used policy. If the requested file pages have not yet been cached and the request amount is larger than the pre-allocated memory space, the read/write calls will go directly to the file system. The high water mark of a cache page is used to flush only the dirty data so that an entire page does not always have to be flushed. In addition, cache pages are examined and coalesced if possible to reduce the number of write calls. For file systems that do not provide consistency automatically, such as NFS, we use the approach in ROMIO that wraps the byte-range file locks around each read and write call to bypass the potentially incoherent file system cache (Thakur, Gropp, and Lusk 1999).

4 Experimental Results

The performance evaluation was done on the IBM SP machine at San Diego Supercomputing Center. The IBM SP contains 144 Symmetric Multiprocessing (SMP) compute nodes where each node is an eight-processor shared-memory machine. We use the IBM GPFS file system to store the files. The peak performance of the GPFS is 2.1 GB per second for reads and 1 GB per second for writes. I/O bandwidth peaks at 20 compute nodes. The default file system block size on the GPFS is 256 KB and is also used as the cache page size in the cooperative cache. We present results of three benchmarks: a sliding-window access pattern, NASA's BTIO benchmark, and the FLASH I/O benchmark.

4.1 Sliding-Window Benchmark

To simulate a repeated file access pattern that potentially causes incoherent caching, we constructed the sliding-window I/O pattern depicted in Figure 4(a). The inner loop *j* is the core of sliding-window operation and the outer loop *i* indicates the number of file segments. In each inner loop, a read-modify-write operation on two file pages is performed by each process. A different file segment is accessed as the outer loop is incremented. In the sliding-window access pattern, every process is able to read and write the data modified by all other processes. We compare this with byte-range file locking. This comparison is made under the assumption that cache coherence must be enforced, so a byte-range lock wraps each read and write call sliding-window test code. The performance results are presented in Figure 4(b). File sizes used in our tests range from 32 MB to 1 GB. Bandwidth is obtained by dividing the aggregate I/O amount by the execution time measured from file open to close.

The experimental results clearly show much better performance for cooperative caching. Particularly good speedups are observed in the case of 32 compute nodes. Given the nature of repeated data access in the sliding-window pattern, our caching avoids most of the I/O to the underlying file system by keeping data in memory of the MPI processes. In contrast, the byte-range file locking approach in this case suffers from serious lock contention in the common file regions among as many as 32 processes. In addition, the overhead of using the byte-range locking approach also includes the communication costs for transferring data to or from the servers each time an I/O operation is performed since the system caching is essentially disabled. In principle, I/O should perform better if clients transfer the data from each other's memory with less contention than directly from the file servers. Reducing the involvement of the file servers significantly improves the I/O performance, especially with a large number of processes.

4.2 BTIO Benchmark

Developed by the NASA Advanced Supercomputing (NAS) Division, the BTIO benchmark is one component of the NAS Parallel Benchmark suite (NPB-MPI) version 2.4 (Wong and der Wijngaart 2003). 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 as the square root of the number of processors participating in the computation. Figure 5(a) illustrates the BTIO partitioning pattern with 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. In this paper, we only present the performance results for MPI collective I/O, since collective I/O results in the best performance (Fineberg et al. 1996). There are 40 consecutive collec-

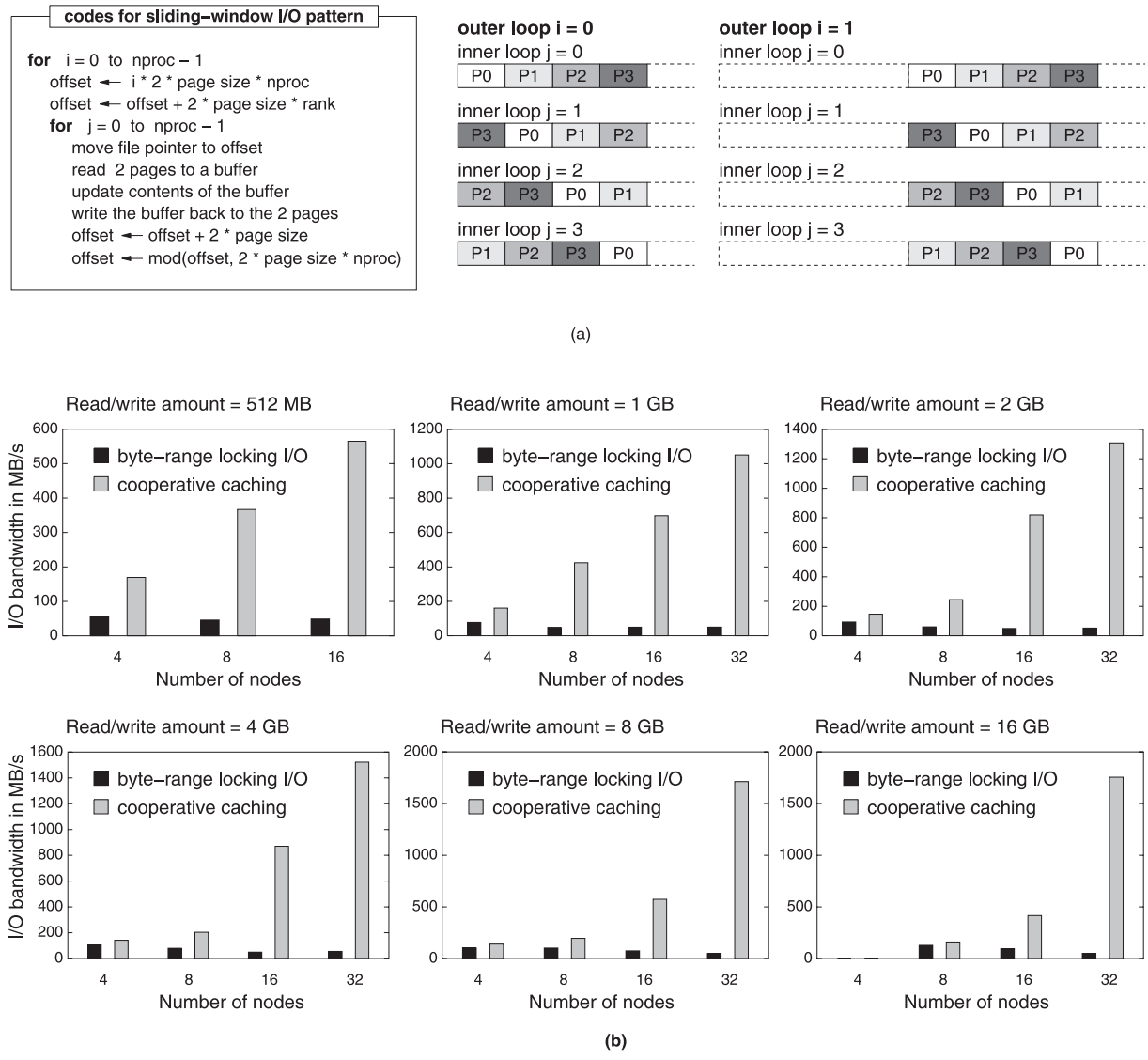

Fig. 4 (a) The sliding-window access patterns. (b) I/O bandwidth for running the sliding-window access pattern.

tive 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 evaluate the A and B classes which generate 800 MB and 3.24 GB of I/O, respectively. Figure 5(b) compares the bandwidth results between cooperative caching and the native approach (without caching and with byte-range locking). In most cases, cooperative caching out-performs the native approach. Cooperative caching can achieve bandwidths near the system peak performance, especially when the number of compute nodes becomes large.

4.3 FLASH I/O Benchmark

The FLASH I/O benchmark suite (Zingale 2001) is the I/O kernel of the FLASH application, a block-structured adaptive mesh hydrodynamics code that solves fully compressible, reactive hydrodynamics equations developed mainly for the study of nuclear flashes on neutron stars and white dwarfs (Fryxell et al. 2000). 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 dimen-

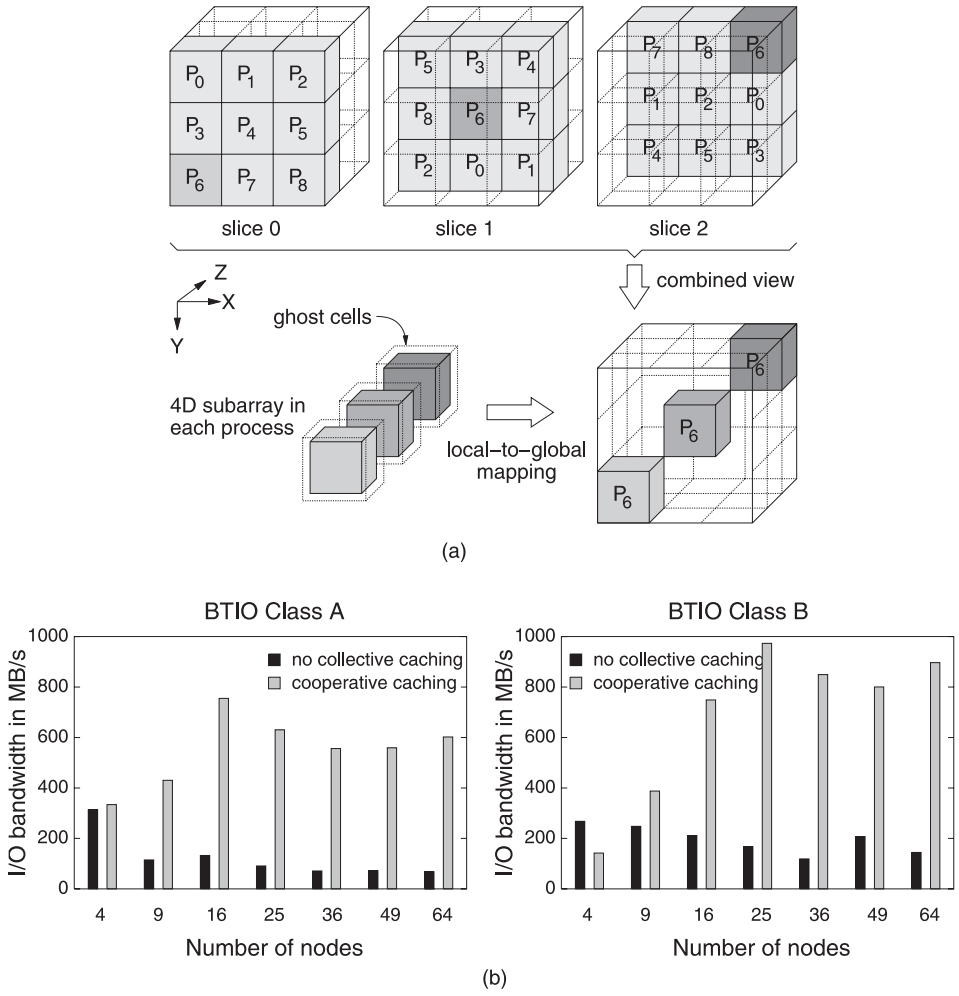

Fig. 5 (a) BTIO data partitioning pattern. The 4D subarray in each process is mapped to the global array in a block-tridiagonal fashion. This example uses 9 processes and highlights the mapping for process P_6. (b) I/O bandwidth results for BTIO benchmark.

sion on both sides to hold information from the neighbors. In this work, we used two block sizes of 8 × 8 × 8 and 16 × 16 × 16. 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, increasing the number of MPI processes linearly increases the aggregate I/O amount. FLASH I/O produces a checkpoint file and two visualization files containing center and corner data. The largest file is the checkpoint file and its I/O dominates the entire benchmark. The I/O is done through HDF5 (HDF Group 2005) which stores data along with its metadata in the same files. HDF5's parallel I/O implementation is built on top of MPI I/O. 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 to 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.

Figure 6 compares the bandwidth results between the I/O implementation with and without cooperative caching. The I/O amount is proportional to the number of compute nodes, ranging from 36 MB to 586 MB for 8 × 8 × 8 arrays and from 286 MB to 4.60 GB for 16 × 16 × 16 arrays. In the 8 × 8 × 8 array size case, we can see that

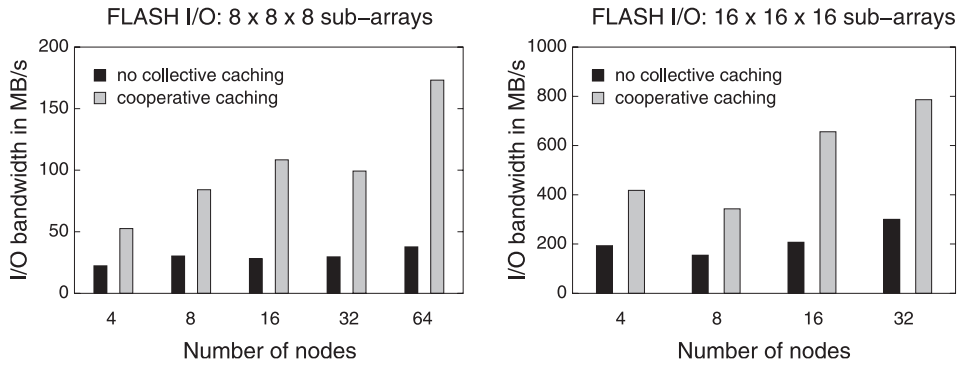

Fig. 6 I/O bandwidth results for FLASH I/O benchmark.

the I/O bandwidth for both implementations is far below the system peak performance. This is because the benchmark uses small I/O amounts and peak performance can really only be achieved with large contiguous I/O requests. Bandwidth improves significantly when the array size increases to $16 \times 16 \times 16$. Without repeating I/O patterns, the I/O performance improvement is the benefit of write behind provided by the cooperative cache.

4.4 Performance Implication

In general, client-side file caching enhances I/O performance for I/O patterns with repeated accesses to the same file regions, and patterns with a large number of small requests. In the former case, caching reduces client–server communication costs. This is exhibited in both the sliding-window and BTIO benchmarks. In the latter, caching accumulates multiple small requests into large requests for better network utilization, also known as write behind and read ahead. Running forty consecutive write operations in the BTIO benchmark indicates that the write behind strategy is beneficial. Similarly, the FLASH I/O benchmark contains only write operations, so write behind is attributed for the performance improvement of the cooperative cache.

Cooperative caching bears several sources of overhead including memory copies between I/O buffers and cache pages, distributed lock management, and communication for accessing remote cache pages. For environments with a relatively slow communication network, such overheads can become significant. Therefore, the performance of cooperative caching may be machine dependent. A parameter that can significantly affect the performance of cooperative caching is the logical file page size. The granularity of the file page size determines the number of local and remote accesses generated by an I/O request. For different file access patterns, a single file page size may not always deliver the same degree of performance improvement. If the I/O pattern consists of frequent and small accesses, a large file page size can increase contention for the same pages. However, if a small file page size is used when the accesses are less frequent and with large I/O amounts, a single large request can cover multiple pages and result in many remote data accesses. In some cases, the appropriate page size can only be fine-tuned by the application users.

5 Conclusions

In general, 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 aggregate 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 three benchmarks in our experiments such that the I/O amount does not overwhelm the memory of the compute nodes. In this paper, we propose cooperative caching as a new user-level client-side file cache design for MPI applications. By moving the caching system closer to the applications, the cache system is aware of the processes that will later access the shared file. We have demonstrated significant improvement over traditional approaches that use either byte-range file locking or system default I/O. In the future, we plan to investigate the effects of file page size and explore possible I/O modes that can further help caches deal with a variety of access patterns. Our current implementation for the I/O thread that uses an infinite loop of calling `MPI_Iprobe()` to detect remote requests. Although this design enables the I/O thread to work on a request as soon as it arrives, it can also waste computational resource when low frequent remote requests are

presented. In the future, we will investigate an alternative design to replace the non-blocking probe with a blocking function such that the I/O thread is only activated when remote requests arrive.

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 NSF through the SDSC under grant ASC980038 using IBM DataStar.

Author Biographies

Wei-keng Liao is a Research Assistant Professor in the Electrical Engineering and Computer Science Department at Northwestern University. He received his Ph.D. in computer and information science from Syracuse University in 1999. His research interests are in the area of high-performance computing, parallel I/O, MPI I/O, parallel data mining, parallel file system, and data management for scientific applications.

Kenin Coloma received his B.Sc. degree in computer engineering from Northwestern University in 2001 and began pursuing his Ph.D. under Professor Alok Choudhary shortly thereafter. Kenin has thus far concentrated on high performance library development and optimization with particular emphasis on I/O. Most of his work has been at the MPI-IO level, actively contributing to the ROMIO implementation.

Alok Choudhary is a professor in the Electrical Engineering and Computer Science Department at Northwestern University. He also holds an appointment with the Kellogg School of Management in the Marketing and Technology Innovation Departments. From 1989 to 1996, he was a faculty in the ECE department at Syracuse University. Alok Choudhary received his Ph.D. from University of Illinois, Urbana-Champaign, in electrical and computer engineering, in 1989, an M.S. from University of Massachusetts, Amherst, in 1986, and B.E. (Hons.) from Birla Institute of Technology and Science, Pilani, India in 1982. Dr. Choudhary was a co-founder of Accelchip Inc. and was its Vice President for Research and Technology from 2000-2002. He received the National Science Foundation's Young Investigator Award in 1993, an IEEE Engineering Foundation award, an IBM Faculty Development award, and an Intel research council award. Choudhary has published more than 300 papers in various journals and conferences. He has also written a book and several book chapters. Choudhary serves on the editorial boards of IEEE Transactions on Parallel and Distributed Systems, Journal of Parallel and Distributed Systems and International Journal of High Performance Computing and Networking. He has served as a consultant to many companies and as well as on their technical advisory boards. He is a Fellow of the IEEE.

Lee Ward is a principal member of technical staff in the scalable systems computing department at Sandia National Laboratories. As an inveterate student of operating systems and file systems, his interests have provided the opportunity to make contributions in high performance, parallel file systems, IO libraries, hierarchical storage management and compute cluster integration/management systems.

References

Bernstein, P., Hadzilacos, V., and Goodman, N. (1987). *Concurrency control and recovery in database systems*, Addison-Wesley, Longman, Reading, MA.

Carns, P., Ligon, W., Ross, R., and Thakur, R. (2000). PVFS: A parallel file system for Linux clusters, in *Proceedings of the Fourth Annual Linux Showcase and Conference*, Atlanta, GA, pp. 317–327.

Coloma, K., Choudhary, A., Liao, W.-K., Ward, L., and Tideman, S. (2005). DACche: Direct access cache system for parallel I/O, in *Proceedings of the International Supercomputer Conference*, Heidelberg, Germany.

Corts, T., Girona, S., and Labarta, J. (1996). PACA: A cooperative file system cache for parallel machines, in *Proceedings of the 2nd International Euro-Par Conference*, Lyon, France, pp. 477–486.

Dahlin, M., Wang, R., Anderson, T., and Patterson, D. (1994). Cooperative caching: Using remote client memory to improve file system performance, in *Proceedings of the First Symposium on Operating System Design and Implementation*, Monterey, CA, pp. 267–280.

del Rosario, J., Brodawekar, R., and Choudhary, A. (1993). Improved parallel input/output via a two-phase run-time access strategy, in *Proceedings of the Workshop on Input/Output in Parallel Computer Systems at IPPS'93*, Newport Beach, CA, pp. 56–70.

Fineberg, S., Wong, P., Nitzberg, B., and Kuszmaul, C. (1996). PMPIO – A portable implementation of MPI-IO, in *Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation*, Washington, DC, pp. 188–195.

Fryxell, B., Olson, K., Ricker, P., Timmes, F. X., Zingale, M., Lamb, D. Q., MacNeice, P., Rosner, R., and Tufo, H. (2000). FLASH: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes, *The Astrophysical Journal Supplement Series*, 131(1): 273–334.

HDF Group (2005). *Hierarchical data format, Version 5* (http://hdf.ncsa.uiuc.edu/HDF5), National Center for Supercomputing Applications.

Huber, J., Elford, C., Reed, D., Chien, A., and Blumenthal, D. (1995). PPFS: A high performance portable file system, in *Proceedings of the Ninth ACM International Conference on Supercomputing*, Barcelona, Spain, pp. 385–394.

IEEE (1996). *1996 (ISO/IEC) [IEEE/ANSI Std 1003.1, 1996 Edition] Information technology—Portable operating system interface (POSIX®)—Part 1: System application: Program interface (API) [C Language]*, pub-IEEE-STD: adr: pub-IEEE-STD.

IEEE (2001). *IEEE Std 1003.1-2001 Standard for Information Technology—Portable operating system interface (POSIX) system interfaces, Issue 6*, pub-IEEE-STD:adr: pub-IEEE-STD. Revision of IEEE Std 1003.1-1996 and IEEE Std 1003.2-1992, Open Group Technical Standard Base Specifications, Issue 6.

Isaila, F., Malpohl, G., Olaru, V., Szeder, G., and Tichy, W. (2004). Integrating collective I/O and cooperative caching into the "Clusterfile" parallel file system, in *Proceedings of the Eighteenth Annual International Conference on Supercomputing*, Sain-Malo, France, pp. 58–67.

Cluster File Systems (2003). Whitepaper:*Lustre: A Scalable, High-Performance File System*, Cluster File Systems, Inc.

Ma, X., Winslett, M., Lee, J., and Yu, S. (2003). Improving MPI-IO output performance with active buffering plus threads, in *Proceedings of the International Parallel and Distributed Processing Symposium*, Nice, France.

MPI Forum (1995). *MPI: A message passing interface standard, Version 1.1* (http://www.mpi-forum.org/docs/docs.html).

MPI Forum (1997). *MPI-2: Extensions to the message passing interface* (http://www.mpi-forum.org/docs/docs.html).

Nagle, D., Serenyi, D., and Matthews, A. (2004). The Panasas ActiveScale storage cluster – Delivering scalable high bandwidth storage, in *Proceedings of the ACM/IEEE Conference on Supercomputing*, Pittsburgh, PA, pp. 53.

Prost, J., Treumann, R., Hedges, R., Jia, B., and Koniges, A. (2001). MPI-IO/GPFS, an optimized implementation of MPI-IO on top of GPFS, in *Proceedings of the ACM/IEEE Conference on Supercomputing*, Denver, Colorado, pp. 17.

Prost, J., Treumann, R., Hedges, R., Koniges, A., and White, A. (2000). Towards a high-performance implementation of MPI-IO on top of GPFS, in *Proceedings of the Sixth International Euro-Par Conference on Parallel Processing*, Munich, Germany, pp. 1253–1262.

Schmuck, F. and Haskin, R. (2002). GPFS: A shared-disk file system for large computing clusters, in *Proceedings of the First Conference on File and Storage Technologies*, Monterey, CA, pp. 231–244.

Tanenbaum, A. and van Steen, M. (2002). *Distributed systems – principles and paradigms*, Prentice Hall, Upper Saddle River, NJ.

Thakur, R., Gropp, W., and Lusk, E. (1999). On implementing MPI-IO portably and with high performance, in *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, Atlanta, GA, pp. 23–32.

Voelker, G., Anderson, E., Kimbrel, T., Feeley, M., Chase, J., Karlin, A., and Levy, H. (1998). Implementing cooperative prefetching and caching in a globally-managed memory system, in *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, Madison, WI, pp. 33–43.

Wong, P. and der Wijngaart, R. (2003). NAS parallel benchmarks I/O Version 2.4, Technical Report NAS-03-002, NASA Ames Research Center, Moffet Field, CA.

Zingale, M. (2001). FLASH I/O benchmark routine – parallel HDF 5 (http://flash.uchicago.edu/~zingale/flash_benchmark_io).