Noncontiguous Locking Techniques for Parallel File Systems

Avery Ching, Wei-keng Liao, and Alok Choudhary
Department of EECS
Northwestern University
Evanston, Illinois 60208
{aching,wkliao,choudhar}@ece.northwestern.edu

Robert Ross
MCS Division
Argonne National Laboratory
Argonne, IL 60439
rross@mcs.anl.gov

Lee Ward
Scalable Computer Systems
Sandia National Laboratories
Albuquerque, NM 87185
lee@sandia.gov

ABSTRACT

Many parallel scientific applications use high-level I/O APIs that offer atomic I/O capabilities. Atomic I/O in current parallel file systems is often slow when multiple processes simultaneously access interleaved, shared files. Current atomic I/O solutions are not optimized for handling noncontiguous access patterns because current locking systems have a fixed file system block-based granularity and do not leverage high-level access pattern information.

In this paper we present a hybrid lock protocol that takes advantage of new list and datatype byte-range lock description techniques to enable high performance atomic I/O operations for these challenging access patterns. We implement our scalable distributed lock manager (DLM) in the PVFS parallel file system and show that these techniques improve locking throughput over a naive noncontiguous locking approach by several orders of magnitude in an array of lock-only tests. Additionally, in two scientific I/O benchmarks, we show the benefits of avoiding false sharing with our byte-range granular DLM when compared against a block-based lock system implementation.

1. INTRODUCTION

Researchers in fusion (GTC [18]), combustion (S3D [30]), molecular dynamics (NAMD [27], Desmond [5], and Blue Matter [14]), astrophysics (FLASH [15]) and many other fields are achieving scientific breakthroughs by using large-scale computing systems to simulate experiments which are difficult to pursue in the physical world. CPU, memory, and network components have made great performance leaps that have enabled new problems to be tackled on these resources. Storage systems, however, have lagged significantly. Because of this disparity, scientists find themselves hamstrung by the I/O system in their ability to store simulation results. Once these results are stored, the relatively slow I/O systems further limit the rate at which results can be analyzed. The addition of computational power and memory means that simulations may be performed at a higher data resolution or with more time steps, which compounds an already difficult situation for the storage system.

I/O system software and middleware is used to mitigate this situation. Parallel file systems provide a way for I/O bandwidth to scale on par with their computational counterparts. One area of progress in I/O has been the addition of the MPI-IO interface in 1997 [24], which provides the same portability and rich access pattern descriptions for parallel I/O as MPI did for parallel computing. Additionally, high-level I/O libraries built on top of the MPI-IO interface, such as HDF5 [16] and parallel netCDF [20], provide programmers with high-level I/O APIs and portable file formats. When scientists can use high-level descriptions for the access pattern of their application, optimizations such as datatype I/O [9] and collective I/O [13] can be applied to significantly improve performance. One implementation of collective I/O, two-phase I/O [32] (unrelated to the two-phase lock protocol), provides good performance in many cases and can help solve the atomicity problem. Two-phase I/O, however, requires that all processes coordinate their I/O and is not suitable for individual I/O and/or unbalanced workloads. Our work can be applied to collective I/O techniques, such as two-phase I/O, although in this paper we target individual I/O operations.

One area of significant difficulty for the I/O library developers is to support atomicity for these high-level I/O APIs. Atomic high-level I/O operations are useful when regions of data in a file are shared by multiple processes, such as in HDF5 where internal metadata in a file is used by all processes to place application data in a consistent manner. The MPI-IO *atomic mode* consistency semantics guarantee sequential consistency for all I/O operations among processes which opened up a file collectively. This mode may be used to implement the type of sharing described above.

Atomicity is even an issue for contiguous I/O operations. For example, one process *P0* may write a contiguous region of data which spans two I/O servers *I0* and *I1*. Another

SC07 November 10-16, 2007, Reno, Nevada, USA

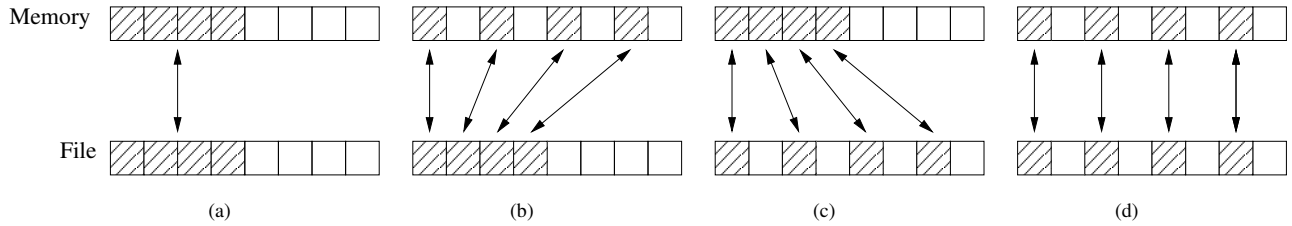

Figure 1: Various I/O access patterns which denote contiguous and noncontiguous descriptions in memory and file.

process *P1* may read the same regions of data. If *P0* writes to *I0* first and *I1* second and *P1* reads from *I1* first and *I0* second, then *P1* may see only part of *P0*'s write, violating atomicity.

The most common technique for implementing atomicity is to use file locking, and most file locking implementations are limited in their ability to describe noncontiguous regions. Unfortunately, noncontiguous access patterns (see Figure 1) are common in scientific applications [2, 11]. Because file locking implementations perform so poorly for these patterns, the MPI-IO atomic mode is rarely used. The lack of an efficient atomicity implementation for high-level I/O APIs, such as MPI-IO, has led many parallel application designers to simply have each process write to its own file and manage this collection of files with scripts and custom post-processing tools. This is a very inefficient solution, and it lowers the scientist's productivity.

Efficient atomic noncontiguous I/O operations could be used for many producer-consumer problems, including real-time visualization of data, and are critical for in-place data manipulation techniques. In addition to giving support to high-level APIs, efficient atomic noncontiguous I/O operations are an important building block in software-level RAID techniques and file system journaling, both of which are growing in importance as we build file systems from ever-larger collections of storage devices.

Several solutions have been proposed to provide atomic I/O for the file system. Most of them use some form of locking for handling atomicity. Solutions in ROMIO [29], the most popular implementation of MPI-IO, include the use of byte-range locking across the entire access pattern and file locking with MPI one-sided communication [19]. While these approaches offer some important benefits, they are only useful within the context of MPI-IO accesses and rely on MPI-2 calls that are not available on many large systems. More general solutions approach the problem at the file system level. Atomicity at the file system level typically uses some fixed size granularity that is a multiple of the file system block size.

In this paper, we propose using a scalable distributed lock manager (DLM) architecture which has true byte-range granularity for handling atomicity within shared files. We present list and datatype locking methods that leverage high-level noncontiguous access pattern information and hybrid two-phase lock protocols that make the best use of our new locking methods. We provide synthetic performance evaluations which test the scalability of our lock methods in non-overlapping and overlapping situations. We also test our ideas against two scientific I/O benchmarks, the S3D I/O benchmark and S3aSim, which show our DLM can achieve near lock-less I/O performance. Additionally, we compare our DLM architecture against a block-based cache and locking Lustre implementation to show how false sharing negatively impacts shared file I/O performance.

Our paper is organized as follows. In Section 2, we discuss previous work for implementing atomic I/O in detail and the basic ideas of our research. In Section 3, we describe how we leverage high-level I/O access pattern information to create efficient lock interfaces. In Section 4, we explain how we combined the ordered, rigorous two-phase lock protocol with an optimistic lock protocol to improve overall lock performance. In Section 5, we discuss the implementation of our DLM at both the client and the server. Additionally, we describe how we translate MPI-IO operations into client lock requests. In Section 6, we present the results of our lock tests, S3D I/O benchmark, and S3aSim in a detailed performance analysis. In Section 7, we summarize this paper and discuss possibilities for future work with our DLM.

2. HISTORY & OUR DLM APPROACH

Most file systems today follow the POSIX standard for I/O [17], which states that a read operation, that can be proven to occur after a write, must return the new data (the entire write should be visible). Further, I/O operations should be "atomic," where they are only seen in their entirety, or not at all. The MPI-IO atomic mode is similar: it guarantees sequential consistency of writes and reads to a group of processes which have collectively opened a file. The POSIX I/O API limits atomicity to contiguous regions of the file. MPI-IO, however, also supports atomic noncontiguous I/O operations.

Parallel file systems today do not have an optimized approach for handling efficient atomic noncontiguous I/O access. IBM's GPFS [31], which is POSIX compliant, has an optimized MPI-IO implementation [28] for improving collective I/O access. It has a global lock manager which hands out "lock tokens" to clients, which helps with optimizing client-side caching. GPFS also employs a special lock division algorithm for acquiring locks on a file which splits the file contiguously among processes. While this may work well for certain cases, it is not generally well suited for interleaved access. For interleaved access, GPFS relies on its data shipping mode for the best performance. Additionally, while GPFS allows byte-range locks, it rounds them to the file system block size, which causes false sharing when writes are not aligned.

Lustre [23] also follows the POSIX standard (except with regard to atime updates and flock()/lockf() system calls) and has a DLM incorporated into its object storage servers

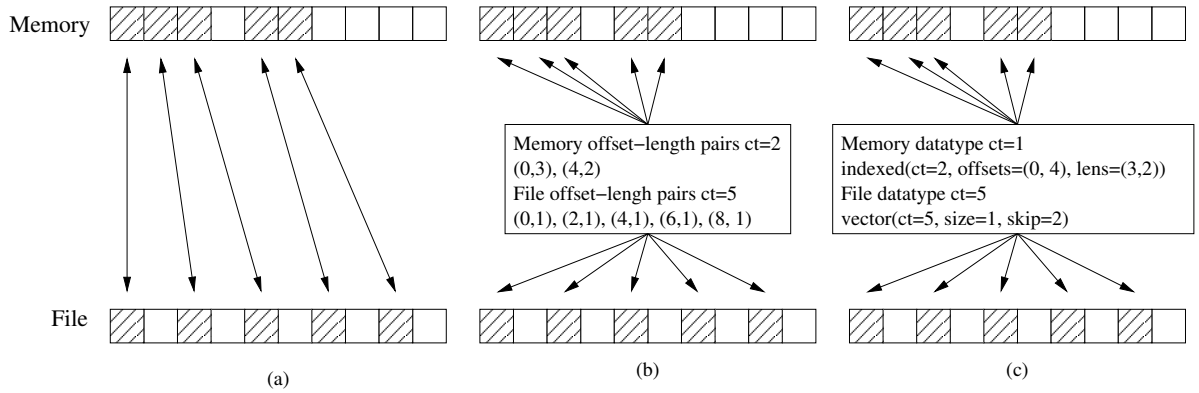

Figure 2: The three lock methods described in this paper: (a) POSIX locking, (b) list locking, and (c) datatype locking.

(OSSs). The OSSs are responsible for locks on the data they manage. The lock granularity on Lustre is the file system page size, where client conflicts are resolved by sending a message to the lock owner to release the page. All cache pages must be flushed before a lock is released. Panasas [25], while supporting file locking, does not appear to have any special support for atomic MPI-IO operations. The Chubby lock service [6], used by Google's Bigtable [7], is a coarse-grain whole-file lock service with a design emphasis on availability and reliability. The Frangipani file system [33] has a single-writer/shared-reader whole-file lock service.

False sharing, which causes serialized I/O operations, occurs when write sizes do not align with the file system lock granularity. In addition to the overhead of serialized I/O operations, file systems that use their lock systems for caching must also flush cache pages before transferring locks among clients. Deadlock avoidance in file systems has been primarily based on the two-phase lock protocol. Our scalable DLM approach seeks to provide efficient noncontiguous lock operations that do not incur any false sharing through true byte-range granular locks. We use high-level access pattern information to create efficient lock APIs for our lock service. Additionally, we have modified the ordered, rigorous two-phase lock protocol to increase performance, while avoiding deadlock (when there are no failures).

3. NEW LOCKING METHODS

Application developers commonly use rich abstractions to describe their data access. In particular, they use high-level I/O libraries such as HDF5 or parallel netCDF, which additionally offer portable file formats. Many high-level I/O libraries, such as those previously mentioned, are built on top of MPI-IO, which is a portable, parallel I/O API. However, some scientific applications are written to directly use MPI-IO. In either case, high-level APIs for I/O can provide opportunities for significant performance improvement. In [9], MPI derived datatype abstractions were efficiently supported by the file system and showed up to several magnitudes of performance improvement over basic POSIX I/O file system operations. We aim to apply these same datatype abstractions to support efficient noncontiguous atomic operations. In following subsections, we describe three different lock methods for implementing atomic noncontiguous I/O.

3.1 POSIX Locking

File systems typically present programmers with the POSIX I/O API. POSIX read and write operations use a pointer to a contiguous region of memory, the current file pointer location, and a count, to access data. We denote the locking counterpart of this interface, which locks a contiguous region of bytes, as *POSIX lock*. The POSIX lock interface can be used to make a noncontiguous access pattern atomic by acquiring all locks to the necessary file regions before doing any I/O as shown in Figure 2a. While POSIX locking can provide atomicity guarantees for noncontiguous I/O access patterns, it has several important drawbacks. The number of lock requests to the lock servers is at least equal to the number of noncontiguous file regions, which creates a significant request processing overhead. When lock regions span multiple lock servers, the number of lock requests increases further due to "splitting" the locks on lock server boundaries.

3.2 List Locking

Using list-based descriptions is a technique that has been used for I/O [8] and in a prototype single lock server [1]. The concept of list I/O (illustrated in Figure 2b) extends the POSIX I/O API to specify multiple noncontiguous regions in both memory and file. Using this technique for locking provides a way to take advantage of the high-level I/O information available from MPI derived datatypes (i.e. we can enumerate the regions).

Some I/O access patterns may have a large number of noncontiguous file regions. We split up list lock requests to the lock servers at every 64 noncontiguous regions since requests should not be arbitrarily large. When application programmers perform unstructured atomic data access, lists of offsets and lengths are a concise way to describe the access patterns to the lock servers. However, offset and length pairs do not concisely capture structured access patterns. When noncontiguous regions have less than 16 bytes, the list locking description with offsets and lengths exceeds the actual data amount being locked.

3.3 Datatype Locking

Many simulation applications use multi-dimensional arrays to model scientific events, such as protein folding, combustion, or fusion. It is common that the I/O accesses in

these data sets, for instance writing one variable for every cell, is regular and structured. Structured data access can be concisely described with a derived datatype. When structured data access requires atomicity, we can use the derived datatype concept with our DLM. The datatype access pattern, shown in Figure 2c, consists of a tree of datatypes as opposed to the offset and length pairs used in list locking. When moving the access pattern description across a network, datatype locking reduces network traffic and the number of lock requests to the lock servers. The lock servers unravel the derived datatypes to determine which locks they are responsible for. If the lock servers were to lack significant processing capabilities, this discovery process could outweigh the benefit of reduced lock requests and network traffic. Additionally, when presented with access patterns containing no regularity, datatype locking breaks down into list locking.

4. HYBRID LOCK PROTOCOLS

Deadlock is always a potential problem when multiple locks are acquired and then released. A variant of the well known two-phase lock protocol [3], rigorous two-phase locking, serializes the order in which operations complete while allowing parallelism on nonconflicting regions. Adding order to the locks acquired in the rigorous two-phase lock protocol eliminates the possibility of deadlock when there are no failures by lock system participants. An ordered, rigorous two-phase lock approach is a good match for noncontiguous file system operations since an order can be imposed based on file offsets. The ordered, rigorous two-phase lock protocol separates the operations into two phases: a growing phase and a shrinking phase. During the growing phase, locks must be acquired in a monotonically increasing order. After doing the necessary work on the locked regions, the client enters the shrinking phase to release all the locks it is holding. During the growing phase, no locks that have been acquired in-order can be released. Similarly, during the shrinking phase, no locks can be acquired. If there are several clients all waiting on various locks, once the "highest" one in the ordering is released, another client will be able to make progress. This client will possibly acquire more locks (assuming all other clients are waiting on other locks) and then release its locks, which allows another client to proceed. This strategy eliminates the possibility of deadlock assuming that clients and lock servers do not fail. For brevity, in the rest of the paper, we refer to the ordered, rigorous two-phase lock protocol as simply the two-phase lock protocol. To improve upon the performance of the two-phase lock protocol, we propose using an optimization that will significantly enhance I/O performance based on the knowledge that it is atypical for a programmer will overwrite their own data. Our optimization is an optimistic lock protocol where clients try to acquire their locks from all lock servers and then release locks that are out-of-order. A simple example would be that client A wants to acquire locks on offset-length pairs (0, 2), (4, 2), (6, 2), and (8, 2). Offset-length pairs (0, 2) and (4, 2), are on lock server 0. Offset-length pairs (6, 2) and (8, 2) are on lock server 1. Client A optimistically tries to acquire all locks from both lock servers simultaneously. Client A waits for the responses from both lock servers and then revokes locks which are out-of-order. If client A has received locks with the offset-length pair (0, 2) from lock server 0 and offset-length pairs (6, 2), (8, 2) from lock server 1, it must release (6, 2) and (8, 2) before deciding whether to retry the optimistic lock protocol or use the two-phase lock protocol.

Incorporating the partial use of an optimistic lock protocol is very important to achieving maximum performance from our optimized lock methods. If a client must in-order acquire every lock through all its noncontiguous file regions, the communication and processing overhead would be tremendous, as is demonstrated in Section 6 with POSIX locking. Below we discuss two combinations of the two-phase and optimistic lock protocols, which helps us to achieve better locking performance in nearly all I/O access patterns.

4.1 One-try Lock Protocol

Using only the optimistic lock protocol to acquire locks might cause a set of clients to end up in a state of livelock, where no locking progress is made. In our *one-try* lock protocol, a client first tries the optimistic lock protocol to acquire locks. If it does not acquire all its locks, it releases the out-of-order locks and then reverts to the two-phase lock protocol of acquiring locks in-order. Most I/O access patterns are not overlapping and therefore benefit from the ability to acquire locks from all servers simultaneously.

4.2 Alt-try Lock Protocol

While there are an endless number of ways to combine optimistic and two-phase lock protocols, we felt that alternating protocols would ensure that progress is made while providing ample opportunity for optimistically acquiring locks. The *alt-try* lock protocol first optimistically tries to acquire all locks, releasing those which are out-of-order. Then it uses the two-phase lock protocol to get its next lock. Upon successfully acquiring its next lock, it again tries to optimistically acquire all locks, releasing those which are out-of-order. This alternating strategy repeats until all locks have been acquired.

5. DLM IMPLEMENTATION

Our DLM implementation has two major components: clients and lock servers. Both components are integrated into the file system. We chose to implement our DLM into the PVFS file system although the ideas could certainly be ported to other parallel file systems or written into a standalone DLM package. This design choice was made for a variety of reasons. As other distributed file systems with atomic capabilities have noted [23], separating the DLM from the file system makes handling failure significantly more difficult. The I/O servers and lock servers may lose communication. If clients lose communication with either the lock server or the I/O server, coordination issues could cause non-atomic behavior. Another reason for the integration is that lock requests can use the same server mapping as I/O requests. Both lock and I/O bandwidths scale as the number of servers a file stripes across is increased. From now on, we refer to I/O servers and lock servers as simply "servers," since they are integrated as one component. PVFS was chosen since it has derived datatype support, which provided a starting point for implementing datatype locking. When PVFS clients access the file system through the MPI-IO interface, they directly interact with the PVFS servers (bypassing the Linux buffer cache on the client). PVFS clients can request byte-granular regions of data from the PVFS servers. Similarly, client lock requests can access actual byte

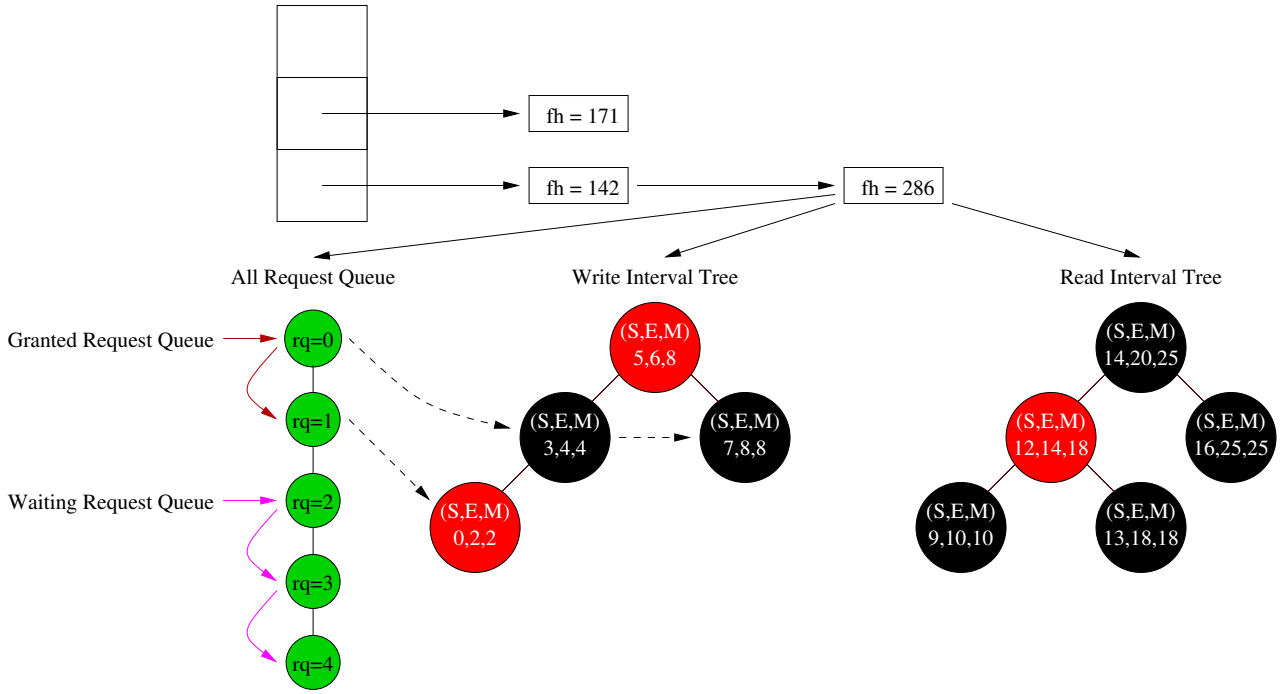

Figure 3: Lock server architecture.

ranges and are not rounded to a sector or file system block size as in other file systems. Therefore, no false sharing is possible with our lock system. Although other file systems may align locks requests to system block sizes which may cause false sharing, they can still benefit from high-level access pattern information and use our lock methods and lock protocols to reduce the number of overall lock requests to the lock system. Section 5.1 and Section 5.2, respectively, describe the client and the server components in detail.

In order to be able to make our DLM implementation usable for MPI-IO applications or other I/O libraries built on top of MPI-IO, we implemented the necessary procedures to convert MPI-IO calls into the appropriate lock calls. This work is described briefly in Section 5.3.

5.1 Client

Our client component is a state machine in PVFS. It begins a lock request by calculating which servers to access, then uses a lock protocol which is chosen at runtime. It supports all three lock protocols (two-phase, one-try and alt-try). All lock protocols are implemented with the two basic lock rounds: optimistic and two-phase. A heap which contains *last_abs_offset* and *next_abs_offset* from each server is used to figure out which servers need to revoke locks and which server has the next in-order lock.

The optimistic round begins when the client attempts to get all locks at once by sending all its lock requests to all the servers involved. It waits for all the replies, which include the last absolute offset locked (*last_abs_offset*) and the absolute offset of the next lock it is waiting on (*next_abs_offset*), and puts this data into the heap. Then the client retrieves the *min(next_abs_offset)* from the heap. If any server has a *last_abs_offset* greater than the *min(next_abs_offset)*, the client must make a request to the relevant servers to revoke locks up to the *min(next_abs_offset)*. At this point, the round is complete.

The two-phase round begins when a client makes a single lock request to the server which has *min(next_abs_offset)* of the heap. The lock request will try to get the locks from *min(next_abs_offset)* to the next *min(next_abs_offset)* of the heap. At this point the server will only reply when it has acquired the necessary locks requested (the client is blocking during this time). Once the client receives the server's reply with *last_abs_offset* and *next_abs_offset*, the client updates the heap and the two-phase round is complete.

The client implements the two-phase, one-try, and alt-try lock protocols described in Section 4 using a combination of these two basic lock rounds. The two-phase lock protocol continually uses the two-phase round. The one-try lock protocol begins with an optimistic round followed by two-phase rounds (if necessary) to fulfill the rest of the lock operation. The alt-try lock protocol begins with an optimistic round, followed by a two-phase round, and continues to alternate lock rounds until the lock operation is satisfied.

Every lock request stores a lock request number for each server involved in the operation. A release request is implemented by sending every server involved in the lock operation the relevant lock request number.

5.2 Server

Servers are not aware of the client lock protocols; they only process acquire (nonblocking or blocking) and release (full or partial) requests. The server uses a variety of data structures for fast lock operations as shown in Figure 3. Files which have any locks are in a hash table in each server for fast lookup. Each file has two interval trees associated with it: a write tree and a read tree. The interval trees provide $O(lg(n))$ algorithmic inserts and deletes, allows easy lookups

for conflicting locks, and are balanced. Each file also has a queue which contains all requests (*all_req_queue*), a queue of waiting requests (*wait_req_queue*), and a red-black tree of granted requests (*granted_req_queue*). The *wait_req_queue* may have lock requests that are blocking or nonblocking. When the server tries to add locks for lock requests in the *wait_req_queue*, it will ignore the lock requests with nonblocking tags.

For a nonblocking acquire request, the server tries to grant as many locks as possible before it overlaps another lock. If the lock request is a read operation, the server checks the write tree and inserts the lock in the read tree if no conflicts are found. If the lock request is a write operation, the server checks the write tree and the read tree for conflicts before inserting locks in the write tree. This implementation provides byte-granular single-writer/shared-reader lock semantics. All locks for a lock request are also chained in a linked list for fast removal. If the lock request has been granted, it is added to the *granted_req_queue*, otherwise it is added to the *wait_req_queue* with a nonblocking tag. Then the server returns *last_abs_offset* and *next_abs_offset* to the client immediately. For a blocking acquire request with a specified final offset, the server again tries to grant as many locks as possible before it overlaps another lock. If the lock request reaches the desired offset, the server returns *last_abs_offset* and *next_abs_offset* to the client. If the lock request is completely finished is added to the *granted_req_queue*. Otherwise, it is added to the *wait_req_queue* with a nonblocking tag. If the lock request does not reach the desired offset, the request is also entered into the *wait_req_queue*, but with a blocking tag. The nonblocking requests in the *wait_req_queue* either add locks or remove locks as specified by later lock requests. The blocking requests in the *wait_req_queue* are checked to see if any locks can be added when locks from other lock requests are released.

Release requests may be full or partial. When the optimistic round is used by a client, partial release requests only remove locks up to a particular absolute offset. A full release typically occurs only after a client has completed all I/O operations protected by its locks. The lock server releases locks by looking up the lock request number in the *wait_req_queue* or *granted_req_queue* and removes the relevant locks using the linked list chain while keeping the interval tree balanced. After the locks are freed, the server returns completion to the client and immediately examines the *wait_req_queue* for lock requests with the blocking tag for servicing. If any of the lock requests in *wait_req_queue* completes, the server notifies the relevant clients.

5.3 MPI-IO Implementation

In order to leverage our scalable DLM implementation for MPI-IO and high-level I/O APIs, we modified the PVFS device driver in ROMIO, which was developed at Argonne National Laboratories. ROMIO supports many parallel file systems through its abstract device interface for I/O (ADIO). Each file system has its own ADIO device driver. In the PVFS device driver we convert MPI file types into PVFS derived datatypes for datatype locking, offset and length pairs for list locking, and contiguous regions for POSIX locking. Additionally, we added several new hints to ROMIO for enabling the various lock methods and lock protocols at runtime.

6. PERFORMANCE EVALUATION

We evaluated the performance of our DLM implementation on the Feynman cluster at Sandia National Laboratories. Feynman, composed of Europa nodes, Ganymede nodes, and I/O nodes, has a total of 371 computers with about 160 nodes available at the time of testing. In order to keep our testing as homogeneous as possible, we only used the Ganymede nodes. The Ganymede nodes are dual 2.4 GHz Pentium-4 Xeon CPUs with 2 GBytes RDRAM and 15 GByte Serial ATA hard drives. They are connected with a Myrinet-2000 network, have access to a production Lustre volume, and use the Redhat Linux Enterprise 2.6.9 operating system. Since each computer has dual CPUs, we used 2 compute processes per node in all our tests.

We configured our PVFS 2.5.1 file system on up to 32 computers as servers, with one computer also handling metadata responsibilities. On tests where we compare against Lustre, we used 16 servers. We kept the default 64 KByte strip size and other default parameters. Compute nodes access the servers through IP over Myrinet. Our Lustre 1.4.7 test directory was configured to use 16 OSTs with a 64 KByte stripe size to match the PVFS configuration. The Lustre lock granularity is aligned to the file system block size for caching reasons and cannot be changed. The Lustre storage nodes (OSSs) are on Infiniband interconnect and connect to the compute nodes via Myrinet to Infiniband routers.

As our DLM uses true byte-range locking, it is not prone to false sharing. In order to understand how this elimination of false sharing would affect performance, we compared our approach to the block-based locking implementation on the Lustre file system on some real-world benchmarks including S3D, a combustion code from Sandia National Laboratories, and S3aSim, a parallel sequence-search algorithm simulator developed at Northwestern University. While, the two systems are not directly compared since they use different hardware, we tried to compare overall trends of how false sharing affects performance. While Lustre has excellent performance in the file-per-process model, shared file performance has been shown to be inefficient due to locking and cache swapping overheads [21]. Lustre does not support atomicity at the MPI-IO level; it is supporting the weaker POSIX consistency semantic which only guarantees atomicity for POSIX I/O operations. Although this is a weaker consistency semantic, the overheads of false sharing are still apparent in our benchmarks. Our initial tests runs with Lustre revealed extremely poor performance (less than 1 MByte / sec) from the MPI-IO data sieving optimization [32] (most likely due to the locking and caching overheads). We set hints in ROMIO to turn off data sieving for noncontiguous data access, which improved performance by an order of magnitude in most cases.

We begin our performance evaluation with a series of lock tests to demonstrate the scalability of our DLM and how it efficiently deals with lock contention in Section 6.1. We continue our analysis with two application based I/O benchmarks. In Section 6.2, we test our DLM in structured data access with the S3D combustion I/O benchmark. In Section 6.3, we demonstrate the performance of our DLM in unstructured data access. Each data point was averaged over 3 runs.

We use numerous lock methods in our benchmarks and define them as follows. *Lustre* refers to the Lustre file system and its block-based caching and locking. *no-lock* refers

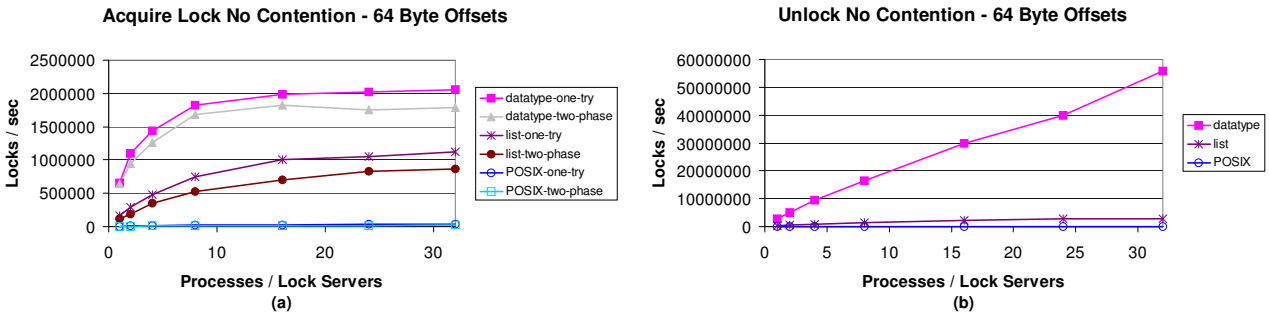

Figure 4: Lock tests without contention: (a) acquire and (b) unlock. Each client accesses locks within an 8 MB range.

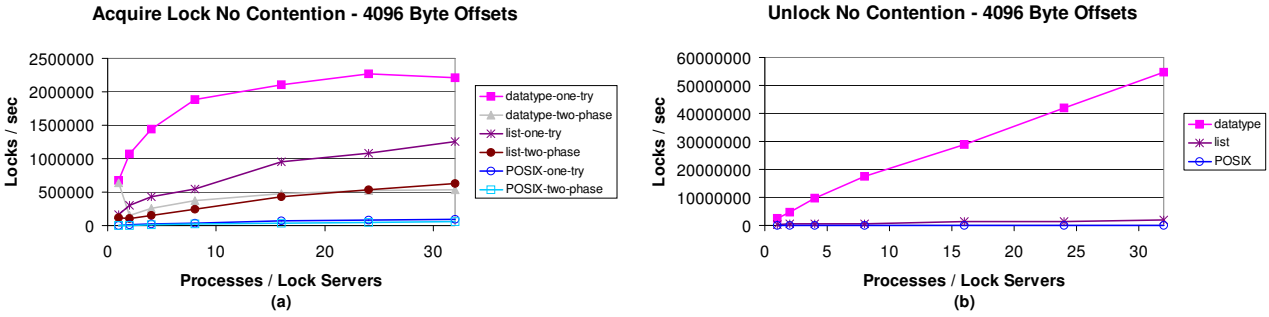

Figure 5: Lock tests without contention: (a) acquire and (b) unlock. Each client accesses locks within a 512 MB range.

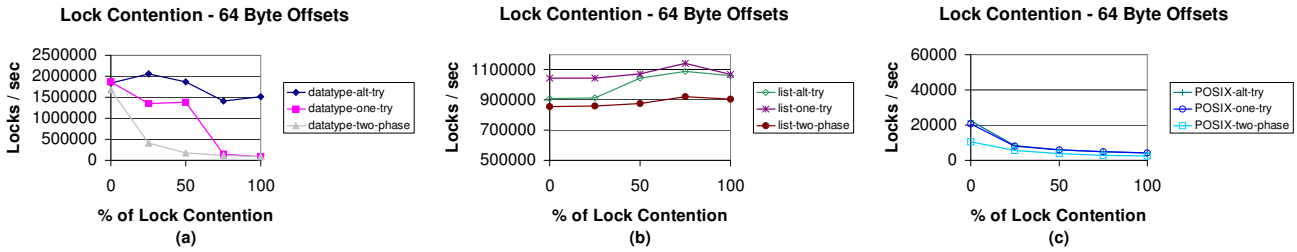

Figure 6: Lock tests with contention: (a) datatype, (b) list, and (c) POSIX. Each client accesses locks within an 8 MB range.

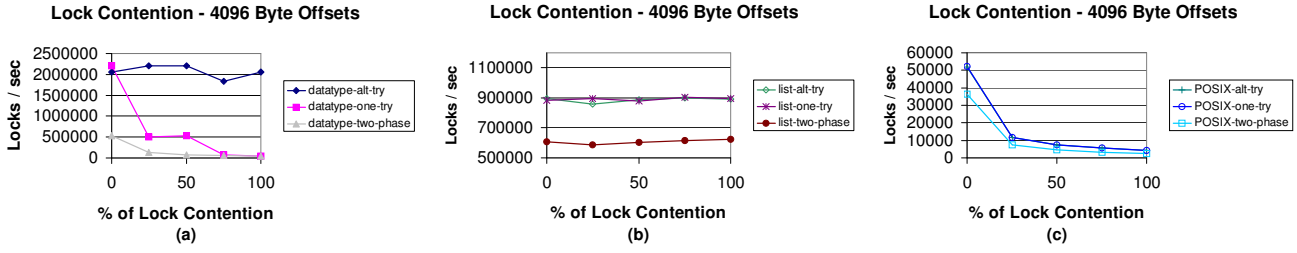

Figure 7: Lock tests with contention: (a) datatype, (b) list, and (c) POSIX. Each client accesses locks within a 512 MB range.

to the PVFS file system only doing I/O. *POSIX-two-phase* refers to using the POSIX locking method with the two-phase lock protocol. *POSIX-one-try* refers to using the POSIX locking method with the one-try lock protocol. *POSIX-alt-try* refers to the using the POSIX locking method with the alt-try lock protocol. Similar references are made to *list-two-phase*, *list-one-try*, *list-alt-try*, *datatype-two-phase*, *datatype-one-try*, and *datatype-alt-try*, respectively.

6.1 Lock Tests

Our lock tests study the locking performance of our DLM directly. We first increased our clients and servers in a 1:1 ratio to see how lock performance scales when given a simple noncontiguous access pattern. Each client attempts to acquire and release 128K locks that are 1 byte long and offset by 64 bytes in Figure 4, and 4096 bytes in Figure 5, respectively. The benchmark begins with measuring the aggregate acquire time (time for all processes to acquire all locks), barriers, and then measures the aggregate release time (time for all process to release all locks). The one-try and alt-try lock protocols have identical performance when the locks are non-overlapping and are represented by the one-try lock protocol in this test. Since we scale up the number of processes with the number of servers in this test, a server has at most 128K locks in its interval tree since locks are equally divided among the servers.

In Figure 4a, acquiring locks increases very rapidly and then stagnates for all methods. Using the two-phase lock protocol is worse for all lock methods due to the communication overhead associated with contacting each server in-order 4 times at 32 processes and servers (the client access pattern spans 8 MBytes compared to a 2 MByte aggregate stripe size with 32 servers). POSIX-one-try is very slow due to the significant number of 128K lock requests to the lock servers. Datatype-one-try reaches a maximum of 2,059,165 locks / sec, a 51 times improvement over POSIX-one-try (40,532 locks / sec), and a 1.8 times improvement over list-one-try (1,123,928 locks / sec). Lock acquiring cannot increase completely linearly in this test case due to datatype processing. In our implementation, all servers which have some locks locally receive the same access pattern description. They must process this access pattern to figure out which locks they are responsible for. As the number of servers is increased, the processing overhead increases since the access pattern processing engine examines each region to sees if it is the owner. While increasing the number of servers reduces the number of locks per server in a given lock request (assuming uniform distribution), the computational processing overhead is not reduced, and therefore limits scalability.

In Figure 4b, we find that unlocking is almost linearly scalable. We only show the unlock bandwidth from each of the three basic lock methods since unlocking is not affected by the acquire lock protocol. In a release request, the server simply finds the matching lock request and removes all locks in a linked list. Since the locks are in an interval tree, keeping the tree balanced is bounded by $O(lg(n))$. However, in most cases the balancing process is $O(1)$, which provides very good performance for unlocking. At its peak, the datatype method unlocks 55,721,183 locks / sec with 32 clients and servers, which is 20 times faster than list locking and almost 3,000 times faster than POSIX locking. The list and POSIX methods are slower since they keep track of more lock requests. Each client using the list method or POSIX method makes 8K unlock requests or 128K unlock requests, respectively. In comparison, clients using the datatype method make a single unlock request.

In Figures 5a and 5b, we increase the range of the noncontiguous access pattern by a factor of 64, which increases the overhead for lock methods using the two-phase lock protocol. Since the two-phase lock protocol requires that locks are acquired in-order, each process acquires 16 locks from a server then must ask the next server for next 16 locks. When the lock offset was 64 bytes, the two-phase lock protocol could acquire 1024 locks from a server, before moving to the next server. In Figure 5a, all lock methods using the two-phase lock protocol fare poorly in comparison to using the one-try lock protocol. Again, since unlocking is not affected by the lock protocol or the lock offset, the results in Figure 5b are nearly identical to Figure 4b.

True lock contention is very rare in scientific applications. As previously mentioned, most lock contention arises from false sharing in other lock systems. Taking this into account, we wanted to test true lock contention to understand how it affected the various lock protocols. In Figures 6 and 7, we kept the locks offset by 64 bytes and 4096 bytes, respectively, and used 32 clients and 32 servers. The locks began with no contention (lined up one after the other) and then overlap each other at 25%, 50%, 75%, and 100% (full overlap) intervals. Each of the graphs has a different scale for clarity. Datatype-alt-try outperforms the other methods with up to 2,062,079 locks / sec in Figure 7a. Using the alt-try lock protocol enables close to full locking bandwidth at all levels of contention. Datatype-one-try and datatype-two-phase significantly drop in lock bandwidth as lock contention reaches 100%, since they must contact the servers multiple times using the slower two-phase protocol. The difference is more pronounced in Figure 7a since the two-phase lock protocol has increased the number of server rounds due to the larger lock offset. List locking is fairly effective in both list-alt-try and list-one-try. List-two-phase does not fair as well, at about a constant $\frac{1}{3}$ drop in performance. POSIX-based lock methods react poorly to increasing lock contention in both the 64 byte and 4096 byte offset cases.

In summary, these lock tests demonstrate how our hybrid lock protocols with list locking and datatype locking provide a large performance increase over a naive POSIX-two-phase method in both overlapping and nonoverlapping cases. In particular, the alt-try lock protocol with the datatype lock method improves performance by an order to two orders of magnitude over the naive locking method.

6.2 S3D I/O Benchmark

S3D is a parallel direct numerical simulation (DNS) solver designed at Sandia National Laboratories [30]. S3D solves the full compressible Navier-Stokes, total energy, species and mass continuity equations coupled with detailed chemistry and is based on a high-order accurate, non-dissipative numerical scheme. S3D writes checkpoint files at periodic intervals which also are used for post-processing in the analysis phase. The three-dimensional Cartesian mesh points of solved variables constitute most of the checkpoint data, which is also in a three-dimensional array. A majority of the checkpoint data is useful during the analysis phase. Since data analysis is an iterative process, the checkpoints are likely to be revisited periodically. Each aggregate checkpoint stores four global arrays, which represent mass, veloc-

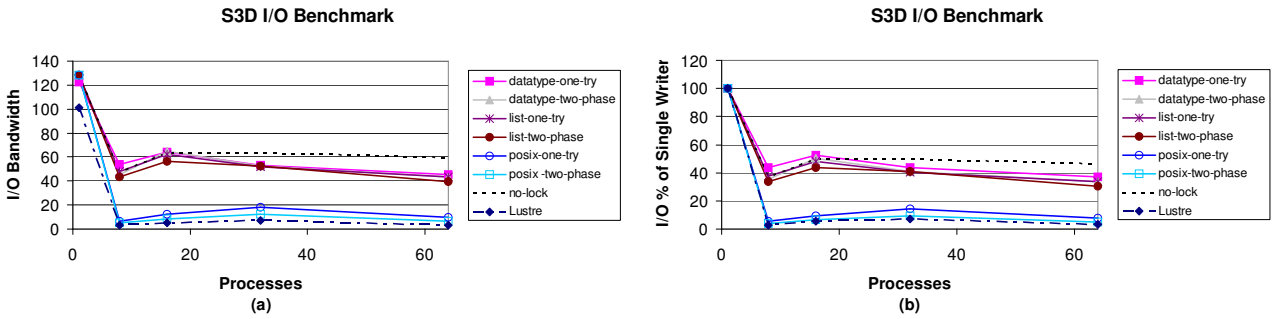

Figure 8: (a) Raw I/O bandwidth. (b) I/O bandwidth as a fraction of single writer I/O time.

ity, pressure, and temperature, respectively. The mass and velocity arrays are four-dimensional and the pressure and temperature arrays are three-dimensional. All four arrays have the same dimensionality for the lowest three spatial dimensions X, Y, and Z. The XYZ dimensions of all arrays are partitioned in the same block-block-block fashion among the MPI processes. The fourth dimension sizes of the mass and velocity data are 11 and 3, respectively, and are not partitioned. The S3D benchmark only performs the checkpoint writes of the S3D code. When a single process is used for checkpointing, its writes are contiguous. As we increase the number of processes for checkpointing, the aggregate data size of approximately 1.19 GBytes remains constant, which makes the access pattern more noncontiguous and reduces the size of each individual write. Smaller writes are a challenging problem for file systems since hard drives prefer large I/O sizes.

The original S3D application uses Fortran I/O, where each process writes its own sub-arrays to an individual file during each checkpoint. While this is typically very fast due to large contiguous I/O calls to non-shared files, it creates a file management problem with an increased number of processes. Additional problems include the requirements that post-processing techniques must access all the individual files and restarts must use the same number of processes. We added an I/O implementation using the MPI-IO API to write the arrays to a shared file in their canonical order. With this change, there is only one file created per checkpoint, no matter how many MPI processes are used, reducing the data management problem. In this test, we try all our lock methods as well as a no-lock method for understanding lock overhead costs. We also compare our single writer normalized results against the Lustre file system to look at false sharing costs (these writes are not aligned to the file system block size). Since the writes are not overlapping, we represent the one-try and alt-try lock protocols with one-try in these tests.

In Figure 8a, we show the overall I/O bandwidth with varying numbers of processes. As expected, as the number of processes increases, overall I/O bandwidth decreases for all methods. Lustre performance begins well with one client since the writes are contiguous and not shared. Then, since the writes are not aligned to the block size, caching and locking overheads reduce performance significantly as the number of processes increases. The POSIX locking method fairs poorly as well due to the large number of lock requests

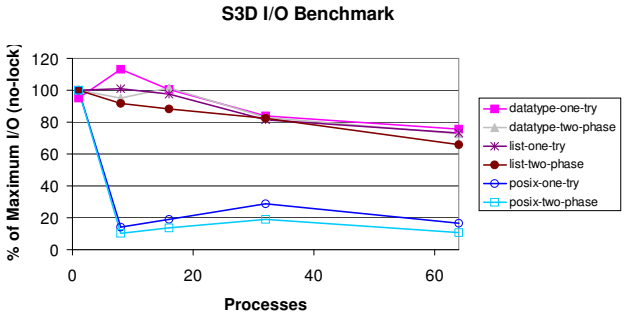

Figure 9: % of I/O bandwidth compared with no locking.

to the server. The list and datatype locking methods fair better with their reduced locking overheads.

In order to make a trend comparison of the Lustre block-based caching and locking methods to our byte-ranged based locking, we normalized the I/O bandwidth as a % of the single write performance in Figure 8b. Most of the lock methods increase slightly from 8 to 16 processes, but fall slightly after that due to the noncontiguous file regions getting smaller and less efficient for the file system. The optimistic locking round in the one-try lock protocol makes a noticeable performance improvement over the two-phase lock protocol. The performance trend for Lustre is a large drop due to its false sharing and associated overheads.

Our final chart in Figure 9 examines the lock overhead of our DLM compared to the I/O bandwidth. POSIX-one-try can, at best, achieve approximately 35% of the no-lock bandwidth due to a large number of lock requests. From 8 to 64 processes, datatype-one-try maintains between 76% to 100% of the maximum I/O bandwidth. List-one-try also keeps I/O bandwidth between 73% to 100% from 8 to 64 processes.

6.3 S3aSim

With the exponential growth of biological sequence databases, parallel sequence-search has recently become a hot topic in computational biology. Tools such as mpiBLAST [12], pioBLAST [22], TurboBlast [4], and many others, are helping scientists understand similarities between newly discovered

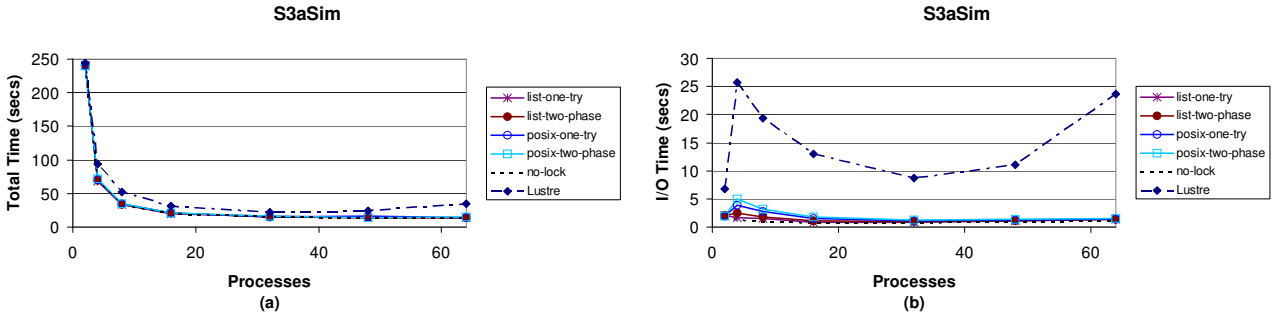

Figure 11: (a) S3aSim total execution time from 2 - 64 processes. (b) S3aSim I/O time from 2 - 64 processes.

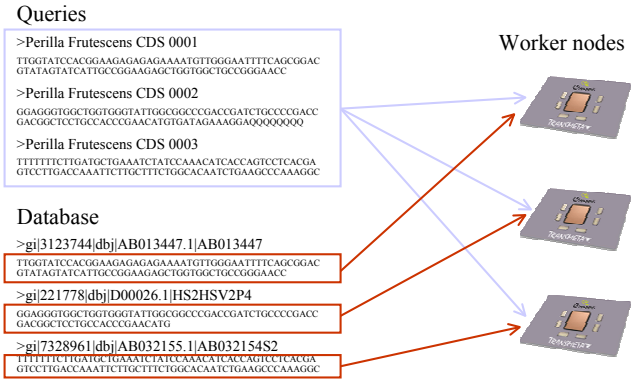

Figure 10: An illustration of database segmentation.

DNA or amino-acid sequences and those in known nucleotide or protein databases. Search results may provide estimates on the evolution distance in phylogeny reconstruction, perform genome alignments, and predict the structure and function of new sequences. Most of these parallel sequence tools have a master-worker architecture and use the database segmentation technique (illustrated in Figure 10) to partition searches between worker processes. In order to provide a powerful tool that could alter input parameters and vary I/O algorithms to optimize future search tools, S3aSim, a sequence similarity search algorithm simulator, was developed [10]. S3aSim uses a variety of input parameters such as the database fragment size, box histograms for the input queries and database sequence sizes, variable computational algorithms, and multiple I/O algorithms.

Our tests were configured based on the NT database characteristics from NCBI [26] with a minimum sequence length of 6 bytes, a maximum sequence length of slightly over 43 MBytes, and a mean sequence length of 4401 bytes. We simulated the search of 20 input queries against 128 database fragments, all with the NT database characteristics. Anywhere between 1000 to 2000 results were pseudo-randomly generated per query and written to file (after the entire query had been completely searched) with MPI_File_write() and then forced to disk with MPI_File_sync(). The results generated are consistent and are not dependent on the number of worker nodes. Each test run produced an aggregate 208 MBytes of output data. We used between 2 - 64 processes in our tests. With only 2 processes, there is a master process and a single worker process, therefore, the worker is writing contiguous data to file. When there are more worker processes writing, the data is noncontiguous and unstructured with varying result sizes and counts. Since the data is unstructured, we do not use the datatype locking method as it breaks down to list locking. Additionally, since the data access is nonoverlapping, the one-try lock protocol and alt-try lock protocol have identical performance, hence we represent both lock protocols with the one-try lock protocol.

In Figure 11a, we look at the scalability of total execution time. The total execution time falls as the number of processes is increased due to the embarrassingly parallel computation. As the number of processes reaches 32, however, the curve flattens out and Lustre starts to increase due to its increasing I/O burden. The Lustre block-based caching and locking has increased false sharing as more processes share file access. The isolated I/O times are shown in Figure 11b.

In Figure 12a, we examine I/O time as a percentage of total execution time. Lustre begins at about 3% when a single worker process is writing to the file. However, as the number of processes increases, the Lustre I/O % increases sharply up to 4 processes and rises up to 68% at 64 processes. The lock methods implemented in our DLM all stay below 11%, rising much slower than Lustre.

In Figure 12b, we check the locking overhead of our atomic operations with respect to I/O times. When there is only a single worker, the lock overhead is practically negligible since it is a single contiguous lock and a large amount of I/O. However, as the number of processes increases, the amount of I/O per worker decreases rapidly and becomes more noncontiguous. For instance, results that may have been contiguous on one worker are now split into two. Therefore, the overall I/O time increases due to smaller I/O requests, which causes the locking overhead to be a smaller % of overall I/O time. In the best case (excluding one worker), list-one-try stays between 70 % to 87% of peak I/O performance, a reasonable overhead for atomicity.

7. CONCLUSION & FUTURE WORK

In this paper, we have presented a novel DLM approach with true byte-range locking using hybrid lock protocols in combination with highly descriptive lock methods to im-

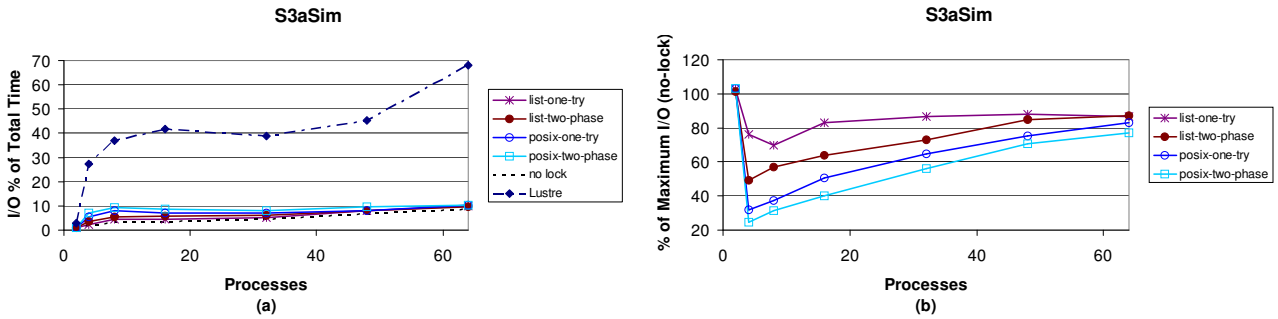

Figure 12: (a) S3aSim I/O time as a % of total execution time from 2 - 64 processes. (b) S3aSim % of I/O bandwidth compared to no locking from 2 - 64 processes.

prove atomic noncontiguous I/O performance. We have shown that this fusion of techniques can improve locking throughput up to between one to two orders of magnitude in performance and maintain a low overhead in achieving atomicity for noncontiguous I/O operations. Additionally, we have shown the benefits of eliminating false sharing with our byte-range granular approach in a comparison with a block-based locking system. Our application benchmarks showed that in most cases, the list or datatype lock methods in conjunction with our optimistic lock protocol exhibited the best performance.

There are many areas where we would like to further explore this work. First of all, an open problem is how to handle failure on clients or servers. While timeout solutions have been proposed to handle client failures, server failures remain an issue. We are investigating the persistent storage of locks as a possible method for resolving this problem. Another area for study is how to best use these lock techniques for supporting the atomic mode for nonblocking, noncontiguous I/O operations efficiently. At present, we are not aware of any support for an atomic mode for nonblocking, noncontiguous I/O operations even though they are part of the MPI-IO specification.

8. ACKNOWLEDGMENTS

We would like to thank our shepherd, Dr. Frank Mueller, for his guidance. This work was supported in part by DOE's SCiDAC program (Scientific Data Management Center) under award number DE-FC02-07ER25808, the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357, the NSF/DARPA ST-HEC program under grant CCF-0444405, NSF HECURA CCF-0621443, the NSF NGS program under grant CNS-0406341, and the DOE HPCSF program.

9. REFERENCES

[1] P. Aarestad, A. Ching, G. Thiruvathukal, and A. Choudhary. Scalable approaches for supporting MPI-IO atomicity. In *Proceedings of the IEEE/ACM International Symposium on Cluster Computing and the Grid*, May 2006.

[2] S. J. Baylor and C. E. Wu. Parallel I/O workload characteristics using Vesta. In *Proceedings of the IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 16–29, April 1995.

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

[4] R. Bjornson, A. Sherman, S. Weston, N. Willard, and J. Wing. TurboBLAST(r): A parallel implementation of BLAST built on the TurboHub. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2002.

[5] K. J. Bowers, E. Chow, H. Xu, R. O. Dror, M. P. Eastwood, B. A. Gregersen, J. L. Klepeis, I. Kolossvary, M. A. Moraes, F. D. Sacerdoti, J. K. Salmon, Y. Shan, and D. E. Shaw. Molecular dynamics—scalable algorithms for molecular dynamics simulations on commodity clusters. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 84, New York, NY, USA, 2006. ACM Press.

[6] M. Burrows. Chubby distributed lock service. In *Proceedings of the 7^{th} Symposium on Operating System Design and Implementation, OSDI'06*, Seattle, WA, November 2006.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7^{th} Symposium on Operating System Design and Implementation, OSDI'06*, pages 205–218, Seattle, WA, November 2006.

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

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

[10] A. Ching, W. Feng, H. Lin, X. Ma, and A. Choudhary. Exploring I/O strategies for parallel sequence database search tools with S3aSim. In *Proceedings of the International Symposium on High Performance Distributed Computing*, June 2006.

[11] P. E. Crandall, R. A. Aydt, A. A. Chien, and D. A. Reed. Input/output characteristics of scalable parallel

applications. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995. IEEE Computer Society Press.

[12] A. Darling, L. Carey, and W. Feng. The design, implementation, and evaluation of mpiBLAST. In *Proceedings of the 4th International Conference on Linux Clusters: The HPC Revolution*, 2003.

[13] J. M. del Rosario, R. Bordawekar, and A. Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *Proceedings of the IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 56–70, Newport Beach, CA, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 31–38.

[14] B. G. Fitch, A. Rayshubskiy, M. Eleftheriou, T. J. C. Ward, M. Giampapa, M. C. Pitman, and R. S. Germain. Molecular dynamics—blue matter: approaching the limits of concurrency for classical molecular dynamics. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 87, New York, NY, USA, 2006. ACM Press.

[15] 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 modeling astrophysical thermonuclear flashes. *Astrophysical Journal Suppliment*, 131:273, 2000.

[16] HDF5 home page. `http://hdf.ncsa.uiuc.edu/HDF5/`.

[17] IEEE/ANSI Std. 1003.1. Portable operating system interface (POSIX)–part 1: System application program interface (API) [C language], 1996 edition.

[18] S. Klasky, S. Ethier, Z. Lin, K. Martins, D. McCune, and R. Samtaney. Grid-based parallel data streaming implemented for the gyrokinetic toroidal code. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 24, Washington, DC, USA, 2003. IEEE Computer Society.

[19] R. Latham, W. Gropp, R. Ross, R. Thakur, and B. Toonen. Implementing MPI-IO atomic mode without file system support. In *Proceedings of the IEEE Conference on Cluster Computing Conference*, September 2005.

[20] J. Li, W. K. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Sigel, B. Gallagher, and M. Zingale. Parallel netcdf: A high-performance scientific I/O interface. In *Proceedings of Supercomputing*, November 2003.

[21] W. K. Liao, A. Ching, K. Coloma, A. Choudhary, and L. Ward. An implementation and evaluation of client-side file caching for MPI-IO. In *Proceedings of the International Parallel & Distributed Processing Symposium*, March 2007.

[22] H. Lin, X. Ma, P. Chandramohan, A. Geist, B.-H. Park, and N. Samatova. Efficient data access for parallel blast. In *Proceedings of 19th International Parallel and Distributed Processing Symposium*, 2005.

[23] Lustre. `http://www.lustre.org`.

[24] Message Passing Interface Forum. MPI-2: Extensions to the message-passing interface, July 1997. `http://www.mpi-forum.org/docs/docs.html`.

[25] D. Nagle, D. Serenyi, and A. Matthews. The panasas activeScale storage cluster - delivering scalable high bandwidth storage. In *Proceedings of the 2004 ACM/IEEE Supercomputing Conference*, November 2004.

[26] NCBI. National center for biotechnology information. http://www.ncbi.nlm.nih.gov/.

[27] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kalé. NAMD: biomolecular simulation on thousands of processors. In *SC '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18, 2002.

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

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

[30] R. Sankaran, E. R. Hawkes, J. H. Chen, T. Lu, and C. K. Law. Direct numerical simulations of turbulent lean premixed combustion. *Journal of Physics: Conference Series*, 46:38–42, 2006.

[31] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the Conference on File and Storage Technologies*, San Jose, CA, January 2002. IBM Almaden Research Center.

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

[33] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: a scalable distributed file system. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 224–237, New York, NY, USA, 1997. ACM Press.