Scalable Approaches for Supporting MPI-IO Atomicity

Peter M. Aarestad, Avery Ching, George K. Thiruvathukal,[*] and Alok N. Choudhary
Electrical Engineering and Computer Science Department
Northwestern University
{aarestad, aching, gkt, choudhar}@ece.northwestern.edu

Abstract

Scalable atomic and parallel access to noncontiguous regions of a file is essential to exploit high performance I/O as required by large-scale applications. Parallel I/O frameworks such as MPI I/O conceptually allow I/O to be defined on regions of a file using derived datatypes. Access to regions of a file can be automatically computed on a per-processor basis using the datatype, resulting in a list of (offset, length) pairs. We describe three approaches for implementing lock serving (whole file, region locking, and byte-range locking) and compare the various approaches using three noncontiguous I/O benchmarks. We present the details of the lock server architecture and describe the implementation of a fully-functional prototype that makes use of a lightweight message passing library and red/black trees.

1 Introduction

When application designers need to implement real-time visualization of data, coherent checkpoint schemes and many other producer-consumer problems, while handling concurrent access to a file by parallel processes, efficient atomic I/O access is required. Atomic I/O can be enforced programmatically (e. g., `MPI_Barrier()` can enforce that only a single process accesses a file at a time), but this approach requires much more work on the behalf of the application designer and does not provide an easy and efficient way to enforce atomic I/O operations.

Large-scale data intensive parallel applications often use MPI-IO natively or through the use of higher level libraries such as pNetCDF [11] and HDF5 [8]. MPI-IO specifies an *atomic mode*, which can be set through the use of `MPI_File_set_atomicity()`. When atomic mode is enabled, MPI-IO will guarantee sequential consistency.

One of the most common ways to enforce sequential consistency is through the use of locks; several examples of implmentations that use locking are mentioned in Section 5. While file locking and byte-range locking are popular strategies, many scientific applications access data using noncontiguous I/O access patterns [3, 5]. Often file locking and byte-range locking force serialized I/O access to processes that may not have overlapping writes and therefore should be able to work in parallel.

We improve I/O concurrency for simultaneous noncontiguous I/O operations by introducing *list locking*. In order to test the various locking strategies on an equal platform, we have developed a general byte-range lock server with communication code that can be built directly into the I/O code to provide exclusive I/O access transparently. Our experiments showed that list locking beat the other methods by over a factor of 8 in some cases and nearly reached ideal performance (non-atomic I/O access).

Section 2 presents an overview of the three different locking approaches in detail. Section 3 describes the software architecture of the lock server and the API exposed to the application developer. Section 4 describes selected I/O benchmarks that we used to test the performance implications of using fine-grained byte-range I/O locking as opposed to the more coarse-grained locking methods of whole-file locking and byte-range locking. Section 5 provides a brief summary of related work. Section 6 presents conclusions and significance of our work and a summary of future research directions.

2 Locking Approaches

We begin by describing the locking mechanisms we used in our tests. When trying to enforce atomic I/O operations, the most common strategies employed are file locking and byte-range locking. We discuss each method in detail. Then we then describe our new method, *list locking*. Figure 1 gives a visual summary of the three methods.

[*] George K. Thiruvathukal is with Loyola University Chicago in the Computer Science department, and has a courtesy appointment at Northwestern University.

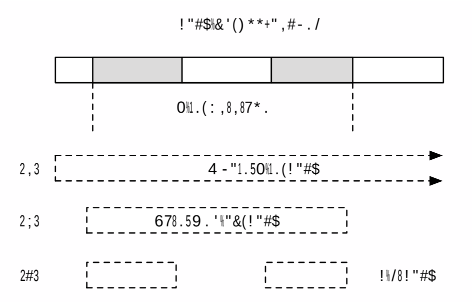

Figure 1. Three different locking approaches to be tested. The shaded region of the file shows the byte ranges used by a particular datatype; (a) shows the extent of a whole-file lock; (b) shows the extent of a byte-range lock, and (c) shows the extent of a list lock.

2.1 File Locking

The easiest way to guarantee exclusive access to an I/O access pattern is to lock an entire file. The first process to request a lock gets, in essence, a lock on the file from its start to its end, no matter how large it grows. Subsequent processes that wish to access the file must wait for the first process to release its exclusive lock, and are then granted exclusive access to the file on a first-come, first-serve basis.

This is obviously the least efficient method of locking a file. Suppose process 1 requests a lock on a file, but only needs to access the first 10 bytes of the file. A second process also wants access to the file, but needs access to a completely different part of the file, for example, bytes 100-200. These two processes could theoretically perform their I/O in parallel, writing their own regions without affecting the other process; however, as long as the first process holds the lock on the file, the second process cannot get any work done. It must wait until the lock is released, acquire the lock itself, and then do the required work. Thus, n atomic I/O operations to the same file would require n sequential operations to complete. The synchronization and serialization overhead of file locking makes it an unattractive option for multiple processes to atomically access the same file.

2.2 Byte-range Locking

Byte-range locking, as seen in Figure 1 (b), changes the granularity of the lock from a file to a range of bytes. When processes lock a single range of bytes instead of an entire file, more concurrent I/O access is possible. Basically, the lock client calculates the beginning and ending bytes of the file access pattern and requests a lock covering the entirety of the access pattern from beginning to end. When other processes make lock requests that do not overlap, they can proceed concurrently.

While byte-range locking generally allows more concurrent access than file locking, if I/O access patterns of multiple processes are interleaved, serialized I/O will result. A simple example of unnecessarily serialized I/O is two processes each writing 10 bytes and then skipping 10 bytes n times. Process 0 starts at byte 0 and process 1 starts at byte 10. While the writes should be able to occur concurrently, the writes will be serialized since the computed bytes ranges used for the byte-range locking strategy are overlapping. This is the same amount of serialization as if we had employed the file locking strategy.

2.3 List Locking

In order to provide the most concurrency possible, we introduce *list locking*. List locking is a method for locking only the file regions which we are using for I/O. The file locking and byte-range locking methods actually provide exclusive access to regions of file that are required for I/O and some regions of file that are not. Since we can describe a locking access pattern identical to the I/O access pattern, list locking can provide the highest level of concurrency possible for atomic I/O operations. For example, many scientific applications use multi-dimensional arrays as data structures. Most such applications use the nested `MPI_Type_vector()` calls to describe strided access to data. By examining the datatype, we can determine only those byte ranges affected by the datatype, and request locks on only those byte ranges. Using this method, we can provide better I/O concurrency to handle the cases where the entire range of two datatypes overlap, but the individual elements in the datatype do not overlap. If each process is granted locks only on those exact byte ranges it is affecting, then other processes can be granted locks on file access patterns that do not overlap those byte ranges.

List locking introduces two complications that were not present before. The first issue we must consider is the actual computation of the byte ranges. This is a decidedly non-trivial issue—MPI derived datatypes can be nested, resulting in highly complex access patterns. The other complication that will arise is the increase in communication that will be required to describe the byte ranges to be locked. While a file lock request would simply require the name of the file to be locked to be transmitted, and a byte range lock would require the file name and the minimum and maximum offset of the datatype, list locks may require tens, hundreds, or perhaps thousands of offset-length pairs to be transmitted from the client to the server. This must be done efficiently to allow lock requests to be transmitted and processed quickly and to reduce the overhead of requesting locks to a minimum.

3 The List Lock Server

Our lock server uses a two-tiered server system as illustrated in Figure 2 There are two actual server applications deployed in our lock server: the client proxy and the server proper. The actual server stores all the state on locks currently held by various clients; the client proxy does not store any state, and acts essentially as a local proxy to the remote lock server. The current implementation of our lock server does not require this two-tiered approach, but as we will explain in Section 6, the client proxy will eventually be able to store information on the age of locks, and will communicate directly with its application clients if the server chooses to remove locks that have been in existence for too long. Our focus, though, was on performance initially, and so we made the decision to leave this feature as future work.

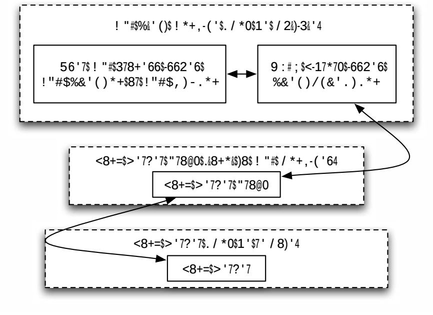

Figure 2. Lock server architecture.

3.1 Software Architecture

The client proxy and server are both written in Java using the Java Development Kit from Sun Microsystems. The use of Java in high-performance computing tasks is considered controversial [9] when it comes to performance; however, our results show that a lightweight server written in Java such as ours can still produce good results. (As an aside, the lock server focuses on networking and data structures, both of which are known to work efficiently in Java unlike floating point, which is known to run slowly as described in the preceding reference.) The locks themselves are stored as a simple byte range associated with a particular file. Determining the ideal method for storage of the locks on the server side proved to be difficult, due to the computational complexities of searching through existing locks in order to add new ones and remove old ones. We settled on using a red-black binary search tree [7] to store the locks associated with each individual file; references to the trees themselves would be stored in a hash table using the name of the file as the key. This allows for $O(m \log y)$-time insertion (where m is the number of requested locks for a particular file, and n is the number of existing locks on a particular file) and better than $O(m \log y)$-time deletion, since we have a direct handle to each object deleted into the tree, thus the actual deletion can be done in $O(1)$ time, with rebalancing taking no more than $O(\log n)$ time on average. Unfortunately, although the Java standard library provides a high-performance hash table (in particular, `java.util.HashMap`), it does not provide a flexible balanced binary search tree suitable for our needs. Eventually, we settled on a simple red-black tree implementation used in a popular modern data structures textbook [6] and provided on their web site at `http://net.datastructures.net`. This implementation proved to be the best of the several that we found in our search. As shown in Figure 3, the comparison function used to store the lock objects in the tree compares the actual interval (range of bytes) represented by the locks; if the whole interval of a lock (request) lies "to the left" of another lock (i.e. the interval's upper bound is strictly less than the other lock interval's lower bound), the lock on the left is strictly less than the one on the right; likewise, a lock "to the right" of another lock (i.e. its lower bound is strictly greater than the other lock's upper bound) is strictly greater than the lock to the left. If any part of two lock intervals overlap, they are considered to be "equal".

Needless to say, interval mathematics is important for list locking and in high-performance scientific/technical computing. There was a proposal—in which co-author Thiruvathukal served as the lead editor—to bring direct support for the concept to Java [1], which would have made Java one of the first modern languages to support interval mathematics. A side effect of our list locking work could be a first step toward integrating interval mathematics with modern data structures.

3.2 Client-Server Communication

The application, client proxy and server use a lightweight communication protocol on top of TCP/IP. The protocol, and associated Java-based implementation, was developed as part of the JHPC class library described in [4] to demonstrate best practices in high-performance Java computing; the source code can be found at `http://www.jhpc.info`. In our original design, all data was transmitted from the sender to the recipient as key-value pairs, with both the key and value represented as strings (in particular, instances of `java.lang.String`), and stored as Message objects on the client and server side. Java transmits strings in an efficient manner using a mod-

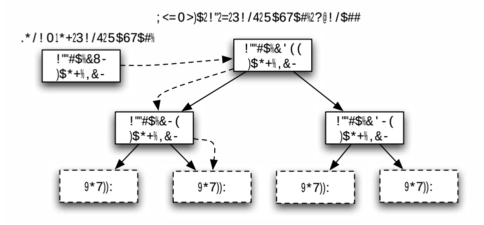

Figure 3. Illustration of lock comparison. The incoming lock request follows the dashed-arrow pattern to ensure that an overlapping request is not already in the tree; it is then added and the tree is rebalanced if necessary.

ified UTF-8 encoding with a 16-bit header denoting the string's length; thus, ASCII strings of length n are transmitted over the wire in $n+2$ bytes. In particular, 32- and 64-bit integers were converted to strings and sent this way. This is actually more efficient for smaller integers, but since we are representing the offsets and lengths of the individual locks as 64-bit integers, the values could potentially become very large. Also, each value would be associated with a separate key, which would lead to massive redundancy in the information being sent. We thus modified the communication protocol to allow arrays of integers (both 32- and 64-bits) to be marshalled in their natural machine representation, which proved to be much more efficient. For example, an array of 10 64-bit integers with a 5-character-long key (such as `'array'`) could then be sent in $7 + (10 * 8) = 87$ bytes. If each 64-bit integer needed a separate 5-character-long key, and each integer was 12 decimal digits long, the transmission would require $10 * (7 + 14) = 210$ bytes.

In order to allow applications written in C (using MPI) to communicate with the client proxy, we developed a C client to communicate with the existing Java version of the communication software. Because the standard C library provides no standard data structures, we used a lightweight data structures library that is distributed as part of the Apt Compiler Toolkit [16], which was designed and developed by one of the authors. We were able to use this library to implement the data structures necessary to handle Message *objects* - of course, C is not an object-oriented language, but the library was written with object orientation in mind, making it useful to translate the object-oriented Java code into functional C code.

3.3 The Lock Server API

The application programmer will ideally be mostly completely unaware of the existence of a lock server. Ideally, the application programmer will simply issue an MPI-IO function call such as `MPI_write()`, and behind the scenes, the MPI and filesystem implementation will communicate with the lock server to acquire the locks, issue the actual file I/O command, and release the locks.

The exposed API leaves the actual computation of locks to the caller. Once the offsets and lengths of all the individual locks are determined by the caller, they are grouped into arrays and passed to the lock server API which handles communication with the lock server. The lock acquisition call is a blocking call, not returning until all requested locks are granted. Once all the locks are successfully acquired, a lock ID is returned to the device driver, which can then assume it has an exclusive lock over the regions of the file it requested. It can then perform its desired list I/O action, and when it completes, it instructs the lock server API to release the locks granted under its lock ID.

3.4 Lock Computation

A separate function, `lock_datatype()`, is called by MPI-IO implementation internally to compute the locks required for a particular MPI datatype. The locks are computed by using the internal ADIO Flatten calls. The code examines the byte-range locks to be locked and coalesces adjacent requests into one single request in order to reduce the amount of communication with the lock server. This allows us to compute the byte-range locks required for any valid MPI data type.

4 Performance Implications

In order to see whether our improvement in I/O concurrency would validate the use of the list locking approach, we ran a series of noncontiguous I/O tests. Section 4.1 discusses our test machine setup. We used a three-dimensional block benchmark from the ROMIO testing suite, and a simulation of the I/O portion of the FLASH code, and a tile reader benchmark. Each test was run under four different scenarios: no locking, file locking, byte-range locking and list locking. We have include the no locking scenario to provide an upper bound on ideal performance. All results were averaged from 3 test runs.

4.1 Machine Configuration

We ran all of our tests on the Jazz cluster at Argonne National Laboratory [2]. The cluster had the following configuration at test time. There are 350 nodes each with a single 2.4 GHz Pentium Xeon processor. 175 of the nodes have 2 GB of RAM each, and the other 175 nodes have 1 GB RAM each. Further, each node has an 80 GB local scratch disk, Myrinet 2000 connections, Fast Ethernet connection among all the nodes, and a 10 TB global working disk with NFS and PVFS. We conducted our experiments over Fast Ethernet due to our MPICH software not recognizing the Myrinet host names. The nodes run Linux kernel 2.4.29-rc2. MPICH2 version 1.0.2p1 was used in all our testing. All our tests used the shared PVFS parallel file system.

4.2 ROMIO Three-Dimensional Block Test

The ROMIO test suite consists of a number of correctness and performance tests. We chose the `coll_perf.c` test from this suite to compare our methods of noncontiguous data access. The `coll_perf.c` test measures the I/O bandwidth for both reading and writing to a file with a file access pattern of a three-dimensional block-distributed array. The three-dimensional array has dimensions 100 x 100 x 100 with an element size of an integer (4 bytes). Eight total tests were performed, with ten runs of each: each of the four locking scenarios was tested with 8, 27, and 64 processors. Table 1 shows the number of locks generated in each locking scenario and each scenario's maximum concurrency (i.e. the maximum number of processors that can run simultaneously). For the 27-processor list lock test, it should be noted that 3 of the processors acquire only 4 locks each instead of 12.

As can be seen in Figure 4, list locking significantly outperforms all other I/O methods in this test. All I/O access is serialized when using file locking. Therefore, when 8 processes are used, 8 I/O operations are performed sequentially. When 64 processes are used, 64 I/O operations are performed sequentially. Whole-file locking results in a great deal of needless blocking, since the datatypes do not overlap in this test. Thus, most processors end up wasting time trying to acquire locks rather than doing I/O. When we do byte-range locking, we avoid some of the blocking and are able to receive and use locks immediately more frequently, resulting in a much lower execution time. Finally, list locking provides us with full concurrency which eliminates all needless blocking; the only effect on performance is caused by the lock processing overhead.

In the 8-processor test, we see that the bandwidth for list locking is about 93% of the results that did not use locking. However, when we move to 27 processors, the relative performance of our list locking suffers a bit, dropping a bandwidth to about 62% of the average achieved without locking. Similarly, in the 64-processor tests, we see that the performance of list locking drops to about 75% of the average achieved without locking. The discrepancy is likely due to the increase in communication when moving from 8 to 64 processors, and the fact that all the processors had to communicate with a single proxy. Distributing the load among several proxies and servers may alleviate this classic bottleneck situation, but even with the single server, our implementation performed admirably. The fact that the performance of 27 processors was not as good may be due to the odd number of processors, resulting in less-than-optimal communication among the processors during computation.

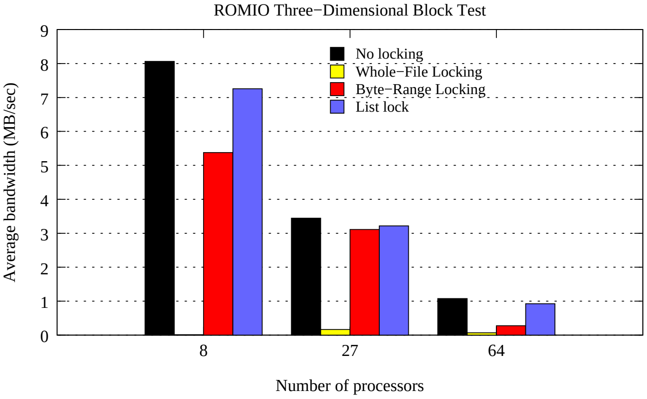

Figure 4. Three-dimensional block test results

4.3 FLASH I/O Simulation

The FLASH code is an adaptive mesh refinement application that solves fully compressible, reactive hydrodynamic equations, developed mainly for the study of nuclear flashes on neutron stars and white dwarfs. The I/O performance for FLASH determines how often checkpointing may be performed, so I/O performance is critical. The actual FLASH code uses HDF5 for writing checkpoints, but the organization of variables in the file is the same in our simulation. The element data in every block on MPI Datatypes. The access pattern of the FLASH code is noncontiguous both in memory and in file, making it a challenging application for parallel I/O systems. The FLASH memory datatype consists of 80 FLASH three-dimensional blocks, or cells in the refined mesh, on each processor. Every block contains an inner data block surrounded by guard cells. Each of these data elements has 24 variables associated with it. Every processor writes these blocks to a file in a manner such that the file appears as the data for variable 0, then the data for variable 1, up to variable 23. Within each variable in file, there exist 80 blocks, each of these blocks containing all the FLASH blocks from every processor. Since every processor writes 80 FLASH blocks to file, as we increase the number of clients, the dataset size increases linearly as well. Every processor adds 7 MBytes to the file, so the dataset ranges between 14 MBytes (2 clients) to 448 MBytes (64 clients). Similarly to the three-dimensional block test, 16 tests were run - four with each

Table 1. Characteristics of each testing scenario for three-dimensional block test

	Number of Processors	Number of Locks per Client	Maximum Concurrent Processes
Whole-File Locking	8	1	1
	27	1	1
	64	1	1
Byte-Range Locking	8	1	4
	27	1	9
	64	1	16
List Lock	8	25	8
	27	12	27
	64	64	64

Table 2. Characteristics of each testing scenario for FLASH I/O simulation

	Number of Processors	Number of Locks per Client	Maximum Concurrent Processes
Whole-File Locking	8	1	1
	16	1	1
	32	1	1
	64	1	1
Byte-Range Locking	8	1	1
	16	1	1
	32	1	1
	64	1	1
List Lock	8	64	8
	16	64	16
	32	64	64
	64	64	64

Table 3. Characteristics of each testing scenario for tile reader benchmark

	Number of Locks per Client	Maximum Concurrent Processes
Whole-File Locking	1 per file	1
Byte-Range Locking	1 per file	2
List Lock	64 per file	2

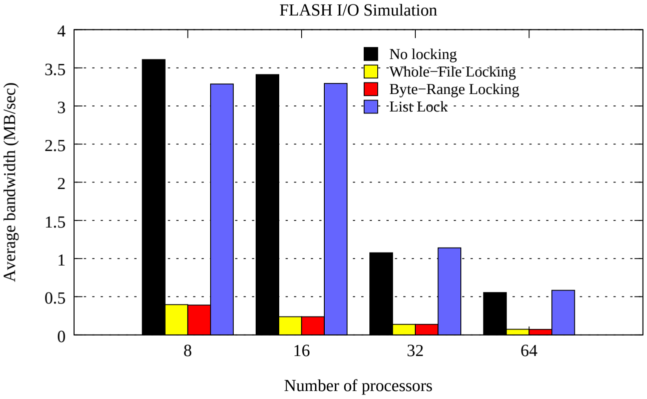

Figure 5. FLASH I/O Simulation results.

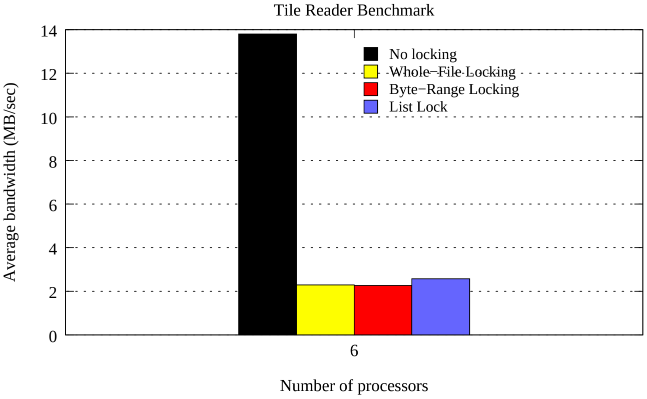

Figure 6. Tile reader benchmark results

of the locking scenarios and each of 8, 16, 32 and 64 processors. Table 2 shows the number of locks generated in each locking scenario and each scenario's maximum concurrency.

Figure 5 shows our results. Again we see that, when there is less contention, list locking performs almost as well as no locking at all. We see again the performance when locking with 8 processors is about 90% of the performance without locking, and list lock performance with 16 processors is about 97% of non-locking performance. Interestingly, the performance when list locking actually exceeds that when not locking with 32 processors by about 6%, and with 64 processors by about 5.5%. A possible explanation for this is that the discipline of achieving locks reduces contention among the processors for access to the file, allowing a more orderly sharing of the file, which would reduce thrashing by the file system.

4.4 Tile Reader Benchmark

Tiled visualization code is used to study the effectiveness of commodity based graphics systems in creating parallel and distributed visualization tools. The amount of detail in current visualization methods requires more than a single desktop monitor can resolve. Using two-dimensional displays to visualize large datasets or real-time simulation is important for high performance applications. Our version of the tiled visualization code, the tile reader benchmark, uses multiple compute nodes, with each compute node taking high-resolution display frames and read- ing only the visualization data necessary for its own display. We use six compute nodes for our testing, which mimics the display size of the full application. The six compute nodes are arranged in a 3×2 matrix of panels, each with a resolution of 1024×768 with 24-bit color. In order to hide the merging of display edges, there is a 270-pixel horizontal overlap and a 128-pixel vertical overlap. Each frame has a file size of about 10.2 MBytes. A set of 100 frames is read for a total of 1.02 GBytes. Five runs were performed using each locking scenario. Table 3 shows the number of locks generated in each locking scenario and each scenario's maximum concurrency.

The results are shown in Figure 6. We can see that, due to a great deal of overlapping byte ranges, there was a great deal of lock contention, resulting in much lower bandwidth versus reading the tiles without locking the file. Still, we show that, even with the large amount of contention, list locking gives us a 12% bandwidth improvement over the more naive locking methods.

5 Related Work

GPFS [13] describes a high-performance filesystem with a novel approach to locking that involves a modification of the whole-file locking mechanism: the first lock requester is granted a lock on the whole file, and subsequent lock requests for the same file cause existing lock requests to be cut in half, similar to the well-known "buddy system" of memory management. Petal [10] describes a distributed file system that locks copies of data blocks before reading or writing to guarantee consistency. NFS version 3 [14] introduced support for file locking, while NFS version 4 [15] also introduced support for byte-range locking. UNIX provides whole-file locking via the file locking function `flock()` and byte-range locking via the function `fcntl()`, as specified in the POSIX standard [12].

6 Conclusions, Significance, and Further Work

Our lock server implementation for MPI datatypes provides a fair comparison of various atomicity strategies. Our tests have shown that list locking is superior to more naive methods of locking in cases where there is no byte-range overlap of derived datatypes (as seen with the FLASH and three-dimensional block tests), and is still somewhat beneficial in cases where there is a great deal of lock contention (as with the tile reader benchmark).

This work has also demonstrated that a robust and scalable lockserver can be developed using object-oriented techniques and Java. The key to success involves knowing when to use Java, avoiding the overhead of starting the Java Virtual Machine within the application itself, being careful with the native class library, and relying upon lightweight approaches for client/server communication.

Future work on this server can be taken in a few different directions. One significant problem not addressed by this implementation is the question of what should be done with *stale* locks; that is, locks that are granted to a process, but are never deleted since the process that requested either dies or otherwise misbehaves. To some extent, this is an implementation detail; however, the current thought is to extend our current architecture by using an approach found in other atomicity mechanisms, such as *pthreads*, wherein operations on a lock can be timed. In our framework, it would be easy to extend the lock concept to allow a time-to-live (TTL) value to be specified, after which the locks would be deleted/released automatically. Performance of the lock server could also be further improved by using an alternative coding of locks, rather than the simple offset/length combination used by our implementation to represent locks.

In terms of list locking itself, many requests—especially those of a computed nature as found in MPI derived datatypes—could be compacted using well known compression techniques (e.g. run-length encoding) or produced by generator (or iterator) expressions. Generator expressions allow you to express a concept—in code—such as the following: Generate a list of locks $(n, n+10)$, for all bytes from $n = 0$ to $n = 1000000$ where $n\%10 == 0$. The advantage of this approach is that the application passes a concise lock request to the server, which expands the request into its present list representation.

Finally, although building a list locking strategy from byte ranges yields promising results, we think it might be possible to do even better by using a block-oriented approach, in which a whole range of bytes could be locked by locking a single bit within a bit vector and eliminate the need to compare overlapping intervals. However, this approach might have the drawback of exposing (unwanted) functionality to the application developer, wherein the block size might need to be specified upon creation of the file itself.

Acknowledgments

This work was funded in part by a grant from the National Science Foundation to Loyola University Chicago (CCF-0444197) and a Cabell Fellowship provided by the Northwestern University McCormick School of Engineering. The authors would also like to acknowledge the advice and input of Kenin Coloma, Wei-keng Liao, and Gokhan Memik, Narayan Desai, Rick Bradshaw Rob Ross, and Benjamín González.

References

[1] *Java Grande Forum Report: Making Java Work for High-End Computing*. http://www.javagrande.org.

[2] *Jazz, the Argonne scalable cluster*. http://www.lcrc.anl.gov/jazz/.

[3] 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.

[4] T. W. Christopher and G. K. Thiruvathukal. *High Performance Java Platform Computing*. Prentice Hall PTR, Upper Saddle Ridge, NJ, 2000.

[5] 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.

[6] M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java, 2nd Edition*. John Wiley and Sons, 2001.

[7] L. J. Guibas and R. Sedgewick. A Dichromatic Framework for Balanced Trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 8–21. IEEE Computer Society, 1978.

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

[9] W. Kahan and J. D. Darcy. *How Java's Floating-Point Hurts Everyone Everywhere*.

[10] E. K. Lee and C. A. Thekkath. Petal: Distributed Virtual Disks. *SIGPLAN Notices*, 31(9):84–92, 1996.

[11] 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 2003*, November 2003.

[12] The Open Group, http://www.unix.org/. *The Single UNIX Specification Version 3, 2004 Edition*, 2004.

[13] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the FAST 2002 Conference on File and Storage Technologies*, San Jose, CA, January 2002. IBM Almaden Research Center.

[14] Sun Microsystems, http://www.faqs.org/rfcs/rfc1813.html. *RFC 1813 - NFS Version 3 Protocol Specification*, 1995.

[15] Sun Microsystems and Network Appliance, http://www.faqs.org/rfcs/rfc3530.html. *RFC 3530 - NFS Version 4 Protocol Specification*, 2003.

[16] G. K. Thiruvathukal and U. Verun. *Apt Compiler Toolkit*. http://www.sourceforge.net/projects/apt.