Interfaces for Coordinated Access in the File System

Sam Lang, Rob Latham, Dries Kimpe, Rob Ross
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
{*slang,robl,dkimpe,rross*}*@mcs.anl.gov*

Abstract

Distributed applications routinely used the file system for coordination of access, and often rely on POSIX consistency semantics or file system lock support for coordination. In this paper we discuss the types of coordination many distributed applications perform, the coordination model they are restricted to using with locks, and introduce an alternative coordination model in the file system. We use extended attribute support in the file system to provide atomic operations on serialization variables, and demonstrate the usefulness of this approach for a number of coordination patterns common to distributed applications.

1. Introduction

The traditional model of a distributed application consists of a set of processes that perform computation on shared state. State is shared using message passing or shared memory, and I/O is performed to the file system for persistent storage of data. In the real world though, this traditional model doesn't always match reality. Because of their autonomy, processes in a distributed application inherently require *coordination*, and while they may use message passing or shared memory to coordinate, they often utilize features of the file system to coordinate amongst processes. In many cases, distributed applications desire some state to persist across application runs, and often utilize the persistent nature of the file system to manage some shared state. Thus, the file system not only manages persistent data, but also acts as a natural place to perform coordination for many distributed applications.

Forms of coordination in the file system include exclusive access to shared data and atomic updates to metadata. These two forms often overlap, as applications often manage their own file metadata within well defined regions of the file (HDF5 stores variable offsets in a header) or utilize atomicity properties of file system metadata interfaces to coordinate on a shared value.

Those applications which do not perform communication directly between processes depend on the file system as the rendezvous manager to enable coordination. For exclusive access to shared data, applications may either assume that interfaces to the file system provide coordination implicitly (sequential consistency of I/O accesses) or may use file system interfaces explicitly to coordinate between processes (using advisory locks or memory mapped files). In both cases, filesystems have traditionally provided coordination through locks, which give a process exclusive access to a file or file regions. With a lock held, a process may perform *critical* accesses to the file (it is expected that accesses *will* overlap), excluding others until the lock is released.

As we will show, locking for coordination is not appropriate for all coordination tasks, such as collaborating on a shared value. Further, distributed lock managers are usually separate from the rest of the file system software, preventing efficient locking implementations based on the coordination patterns of the application. Instead of acquiring and releasing locks, applications may prefer to perform an atomic update of a shared, well defined value. Atomic updates of structured data are not supported by file systems, which traditionally only store unstructured data within a logical namespace.

We propose that the file system provide applications with efficient interfaces to small, structured data values and atomic operation primitives that can be performed on those values. To that end, this paper introduces file system support for persistent atomic operators and presents interfaces to those atomic primitives for applications to build on to perform coordination. We incorporated our atomic operators into the file system using extended attributes on files, allowing us to leverage existing operating system and client-side file system support.

In the next section, we provide background on the types of coordination used by distributed applications, both within the file system and separately. We also discuss some of the approaches file systems have taken to providing locks for exclusive access. In Section 3, we outline some of the desirable properties of atomic operators provided by the filesystem, and define interfaces for a few basic atomic operators. In Section 4, we describe some distributed applications that can utilize atomic operators to implement efficient coordination algorithms, and in Section 5 we show experimental results of those implementations. Finally, Section 6 concludes and discusses possible future directions.

2. Background

2.1. Coordination in the File System

Distributed applications often rely on the file system to provide coordination to shared resources, especially if they do not perform interprocess communication. Locking on files (either the entire file or specific regions) has been the predominant form of coordination, as the file data is most often the primary shared resource. In order to guarantee sequential consistency of individual I/O operations between multiple processes, file systems must implicitly lock a file (or region of a file). To support this, distributed and parallel file systems since the VAX/VMS system have included distributed lock managers [1], [2], requiring sophisticated locking algorithms and client state management. This infrastructure is necessary to support applications accessing *overlapping* file regions, however, not all applications require exclusive access to overlapping file regions. Many distributed applications perform I/O to non-overlapping regions for the majority of their I/O workloads, yet still incur the cost of locking on many parallel file systems, because the file system must provide appropriate consistency semantics independent of the workload [3].

There are a few exceptions to the trend of using lock managers in distributed and parallel file systems. Preslan et al. [4] initially proposed lock extensions to the SCSI interface (called *DLOCK*) allowing storage devices to manage locks directly, and then generalized their approach to allow storage devices to support conditional load and store operations on internal shared memory buffers (known as *DMEP*) [5]. A version was associated with each memory buffer and incremented with each store command, allowing for atomic primitives and other concurrency control mechanisms (such as locks) to be built with the DMEP primitives. The DLOCK/DMEP specifications were implemented by initial versions of the shared disk filesystem GFS [6], [7], but later abandoned in favor of a distributed lock manager [8]. More recently, Ermolinskiy et al. [9] proposed *Minuet*, an optimistic approach to concurrency control in shared disk environments. Minuet adds versioning capabilities to the storage area network (SAN) and provides locking interfaces to the application that allow for *session isolation*. This approach relaxes the *mutual exclusion* property of traditional file system locks and is a better match for many coordination tasks that applications perform. As techniques for improving an application's ability to coordinate within the file system, we note some drawbacks in both approaches. The DMEP approach provides an atomic load and store operation with the goal of implementing locks in a distributed file system, leaving the application with mutual exclusion locks as the only mechanism for coordination. The Minuet approach provides the storage components with the desired consistency semantics of the application but coordination still requires the use of locks. Both approaches require support from hardware vendors, which have traditionally been reluctant to provide anything more than a simple block interface.

Applications with clients competing to write data to overlapping regions (or which need to exclude readers while writing) often do so to small regions to perform coordination, or collaborate on a particular value (such as file header containing the layout information for the rest of the file) [10]. These applications often depend on advisory locks provided by the filesystem, but advisory locks have not been supported by many network and distributed file systems, because of the complexity required to support distributed locking. Applications which require advisory locks have been limited to running on systems where advisory locks are supported, or more commonly, have resorted to using other file system interfaces to gain exclusive access to a file.

For example, a common technique for acquiring exclusive access to a file has been to use the `open` system call with `O_CREAT|O_EXCL` flags, with successful completion indicating that the lock was acquired. Even this technique was not possible everywhere. On NFSv2 (or Linux kernels before 2.6) `O_EXCL` was not atomic and instead, applications were expected to use a combination of the `link()` and `stat()` system calls to a unique "lockfile" [11]. Other applications use the `rename()` system call to atomically update a shared value, by first writing a new value into a temporary file, and then renaming the temporary file to the shared file. These are examples of some of the techniques that have misused the file system interfaces to gain atomicity.

Many applications rely on the file system for collaboration in other ways, such as updating the file

offset on behalf of the application. *Shared file pointer* interfaces are extremely convenient for appending to files or reading from a work queue; I/O patterns that are especially common among data-intensive applications. To support the Map/Reduce programming model [12] the Google File System [13] implements an atomic append operation, a special case of shared file pointers. Some general-purpose file systems have historically offered native support for shared file pointers, including Vesta [14], PFS [15], CFS [16], and SPIFFI [17], but no general purpose distributed file system does so today. Instead distributed applications that need a shared file pointer must update the shared file pointer themselves, and often rely on file system locks to perform the update.

The shared file pointer interfaces defined by MPI-IO have been implemented in ROMIO [18], [19], [20] by means of a hidden file placed on the parallel file system. The hidden file contains the present location of the shared file pointer. When a process initiates a shared file pointer routine, it reads this hidden file for the present value, and writes the new value. In order to ensure atomic access to the hidden file, the `fcntl()` system call is used when available. While support for advisory locks in many file systems has improved, they are by no means standardized. Several parallel file systems, most notably PVFS [21], do not support `fcntl()` locks at all. Lustre [22] supports `fcntl()` locks only if mounted with a special mount option. NFS treats fcntl locks as advisory, not mandatory, and in common configurations will silently let multiple MPI processes concurrently acquire locks, especially if they are on distinct nodes.

2.2. Coordination outside the File System

An alternate approach to implementing shared file pointers based on MPI-2 one-sided RMA operations was proposed in [18]. The MPI-2 one-sided routines provide atomic access to an entire memory region, allowing for a sufficiently clever algorithm built on top of MPI-2 RMA methods.

The RMA approach has one significant drawback. Because shared file pointer operations are independent, the process owning the RMA memory window may stall the locking algorithm's progress, because passive-target RMA operations typically "make progress" only when the target makes other MPI calls[1].

1. A "progress thread" could be added, but on systems with limited thread support (such as the IBM BlueGene/P), a progress thread dedicated to the MPI library has an adverse impact on application performance.

2.3. Coordination using Active Storage

Active storage is a model of computation that moves functions from compute nodes to the file system or storage device. This allows data intensive applications to avoid the cost of moving large amounts of data to the compute nodes, where it may simply be reduced or transformed and shipped back to storage.

In theory, active storage programming models simplify coordination of file accesses, because data accesses become local operations at the storage device, simplifying enforcement of coordination policies.

Thus far, active storage has targeted data intensive computing applications, and has focused on shipping functions that operate primarily on data [12], [23]. Devulapalli et al. [24] propose extensions to the *OSD* standard that would allow for atomic primitives on object-based storage devices, such as *compare-and-swap*, but do not extend the atomic interfaces through the file system to applications.

3. Atomic operations in filesystems

We see today that file systems provide tighter coordination of I/O accesses than is often required by the application. File systems that implement sequential consistency (POSIX file semantics) must implicitly lock regions of a file to gain atomicity. This causes significant overhead even for applications that do not perform overlapping accesses to file regions, because file systems are forced to provide the same semantics for all applications. Attempts to work around the overhead of implicit locks on parallel file systems that support POSIX file semantics have been shown to be non-trivial [3]. Further, distributed applications require atomic access to data in ways that implicit locks do not support. Advisory locks or other obscure techniques to gain atomicity often provide more enforcement than is necessary for the given access pattern, and in many cases hinder desired concurrency.

Instead of forcing the file system to provide the highest semantic guarantees through implicit or advisory locks, we choose to provide the minimum functionality in the file system necessary to *enable* coordination without the file system enforcing mutual exclusion. Our approach is to provide applications with a set of basic atomic operators that allow the application to construct their own coordination models efficiently.

3.1. Properties of Atomic Operators

To allow distributed applications the flexibility to coordinate in different ways, we identify a set of properties desirable for atomic operators:

- **State Variables**. The basic component of an atomic operation is the state that the operation is performed on. To provide atomic operators to the application, we require *typed* state variables exposed by the file system, and interfaces to create, modify and remove state.
- **Persistence**. Applications may have coordination models that require updating values across application runs, or among many applications. Providing atomic operations on *persistent* state variables extends the coordination capabilities of applications. Persistent state can be useful for HPC applications that perform defensive I/O, where checkpoint/restart modes allow an application to discover the state of a run before restarting.
- **Global Namespace**. With persistent state in a distributed environment, applications must be able to uniquely reference state variables and perform atomic operations on them. A global namespace for persistent state provides uniqueness across processes and applications.
- **Per-File Granularity**. In many cases an application coordinates processes around a file. Persistent state variables should exist *on a file*.

With these properties in hand, we identify a few atomic operators that applications can use to provide coordination for two different types of accesses, and describe how we have implemented those operators in the file system in a lightweight manner.

3.2. Fetch and Add

Implementing global shared file pointers naturally leads to the use of an atomic `fetch_and_add` operation, where each update of the shared file pointer increments an offset. In this model, each process of an application performs a `fetch_and_add` operation to some application-specific location, on a per-file basis. The single atomic operation takes as arguments the file, the state variable to perform the operation on, and the value to add to the state variable. The operation returns the value of the variable before the increment.

```
fetched_value =
    fetch_and_add(
        file, variable, add_value)
```

This operation requires an atomic read-modify-write of the variable at the server where the variable is stored.

3.3. Queue Interfaces

Distributed applications that require sequential consistency of I/O accesses must rely on the file system to implicitly lock the file. For those file systems that provide distributed locks to the application, they take on the responsibility of both coordinating between processes, as well as communicating with processes to grant lock requests. The notification mechanisms built into file systems that provide sequential consistency add significant complexity to the overall design of the file system. Our approach is to leverage atomic operators in the file system to perform coordination and enforce fairness of lock requests, and allow the distributed application to communicate between processes to perform notification of locks being granted. In order to enforce fairness, we chose to provide a simple queue interface, where clients requesting a lock place themselves on a queue with a unique identifier (an `enqueue` operation). Clients releasing a lock must remove themselves from the head of the queue (a `dequeue` operation) and notify the client currently at the head that they have been granted the lock. This locking algorithm, which we describe in greater detail in section 4.2, requires only two atomic operations:

```
head_buffer =
    enqueue(file, variable, buffer)

head_buffer = dequeue(file, variable)
```

Both the `enqueue` and `dequeue` operations return the head of the queue, which may be a special `NULL` buffer if the queue is empty. Both operations are performed on a *queue variable*, with variable-length buffers being `enqueued` and `dequeued`. This gives applications the flexibility to use the queue interfaces to meet their needs.

3.4. Operators on Attributes

In order to meet the requirements that we set in section 3.1, we chose to implement the atomic operators using extended attributes. Extended attributes are essentially persistent variables that can be placed on a file. Each extended attribute is given a unique name or key as a string scoped to that file, and the value of the attribute is simply a variable length buffer. This gives us persistent state variables at a per-file granularity. The convention used for extended attributes to define the string key also gives us a global namespace to manage the atomic attributes we choose to support. Leveraging extended attributes allows us to implement support for atomic operators without introducing new file system interfaces or requiring any modifications to the client operating system or file system software. This was an important advantage for us, as modifications to the standard file system interfaces require consensus from standards bodies and

the operating system community, which is often a long and protracted process. Extended attributes do not provide well-defined primitive types, which we require to be able to perform atomic operations on state variables. To obtain primitive types in extended attributes, we embedded the type information in the attribute key:

```
atomic.int.myvariable
```

In this convention, all extended attributes within the `atomic.int.` namespace are known to be integer types. To support atomic operators through extended attributes, we require certain extended attributes to act as *functions* instead of as variables, with support at the file server to recognize an *atomic function*, and to perform the associated atomic operator on the extended attribute variable specified by the operator. In order to accomplish this, we mapped atomic operations to the `atomic.` namespace, and allow atomic operators to be expressed as methods on extended attribute variables within that namespace.

For example, an application may wish to create the following extended attribute:

```
atomic.myapp-run1-shared-pointer
```

This attribute can be created using the standard `setxattr()` system call, and given an initial value of `0`. The attribute would be created on a file for which the application requires shared file pointer access.

```
uint64_t init = 0;
setxattr(
    file,
    "atomic.myapp-run1-shared-pointer",
    &init,
    8,
    XATTR_CREATE);
```

Once created, this extended attribute is visible to all processes using the file system, and each process can perform a `fetch_and_add` operation on that attribute, using the `getxattr()` system call:

```
uint64_t offset;
getxattr(
    filename,
    "atomic.myapp-run1-shared-pointer.fetch_and_add(4096)",
    &offset,
    8);
```

The file server receives the getxattr request, recognizes the fetch_and_add operation being performed on the extended attribute, fetches the attribute, adds 4096 to its value, stores it and returns the fetched value as the response. In this example, each process would be able to perform a 4096 byte access at the offset returned by the getxattr call.

The queue interfaces map to extended attributes in a similar manner. The application might create an atomic *queue* attribute, initialized to zero length:

```
setxattr(
    file,
    "atomic.myapp-run1-lockqueue",
    NULL,
    0,
    XATTR_CREATE);
```

This creates and initializes the queue, at which point processes can request a lock:

```
uint32_t ret;
uint64_t head_rank;
ret = getxattr(
    file,
    "atomic.myapp-run1-lockqueue.enqueue(1)",
    &head_rank,
    8);
```

Here the process specifies its own rank (a value of 1) to be added to the tail of the queue. The return value is used to determine if the queue is empty (used by the locking algorithm further explained in section 4.2). A return value of zero indicates the queue is empty.

Using the extended attributes to perform atomic operations on stored variables allows us to leverage much of the file system infrastructure and interfaces already provided. The approach is not without drawbacks though, and is only meant to demonstrate the usefulness of interfaces that provide flexible coordination mechanisms to the application.

One obvious drawback to using extended attributes in this way is the limited expressiveness of the `getxattr` system call. Describing the atomic operator and its parameters requires serializing the entire operation into a string, which is then given as the extended attribute name. This is only useful for operator parameters that can be encoded into an ASCII string. More problematic is that getxattr is intended to be a read-only operation, so using it to modify state breaks access permission rules.

4. Using Atomic Operators

In this section we describe a few application I/O access patterns that require coordination, and demonstrate the use of our atomic operators to perform that coordination, discussing some of the benefits over other coordination techniques.

4.1. Shared File Pointer Interfaces in MPI

We already know, based on ROMIO's use of a hidden file, that the file system can be a way to provide exclusive access to a shared resource. If we were to store the value of the shared file pointer in a traditional extended attribute, we would be no better off than if we stored the value in a hidden file. We still require a way to perform an atomic fetch and increment. Here atomic operators deliver exactly what ROMIO requires, resulting in a rather simple implementation of

the shared file pointer interfaces for MPI-IO. Figure 1 shows essentially the entire routine, omitting error checking for clarity.

```
snprintf(
    keyval_name, keyval_name_len,
    "atomic.shared-file-pointer.fetch_and_add(%d)",
    incr);
key.buffer = keyval_name;
key.buffer_sz = keyval_name_len;
result.buffer = &result_buf;
result.buffer_sz = sizeof(result_buf);

ret = PVFS_sys_geteattr(&object_ref,
    &credentials, &key, &result, NULL);

*shared_fp = *((int *)result.buffer);
```

Figure 1: Shared file pointer update with atomic operators. In a single call, the attribute `atomic.shared-file-pointer` is incremented by `incr` and the old value returned through `result`.

The atomic operator approach to shared file pointers addresses the shortcomings of both the hidden file approach and the RMA approach. We no longer require the file system to support locks. We also no longer require a progress thread to service RMA requests: the file system metadata server stands ever-ready to service operations.

4.2. Distributed Locks

Due to the high cost of remote memory references, traditional locking algorithms are often not suitable for use in a distributed memory environment. Instead, if a client finds a lock to be taken, it goes to sleep and relies on an external entity (such as a centralized lock manager) to wake it when the lock becomes available. Instead of using a centralized lock manager, it is possible to move the functionality of the lock manager into each of the clients. In this case, the client releasing the lock is responsible for waking the next client in line to obtain the lock. This means that the owner of the lock needs to be able to deduce who, if anyone, should obtain the lock next.

One way of doing this is to use a globally accessible, shared queue which records who has the lock and who is waiting for the lock. To obtain the lock, a client would add itself to the queue. If the queue was previously empty, the lock was obtained. If not, the client goes to sleep knowing that the queue now indicates its desire to obtain the lock. When the lock is released, the owner removes himself from the head of the queue and notifies the client now at the head that they have acquired the lock.

The diagram in Figure 2 shows the locking algorithm. Rank3 currently holds the lock, and is preparing to hand off the lock to Rank1, which moves to the head of the queue. Rank3 sends a notification message to Rank1, notifying it that it has been granted the lock.

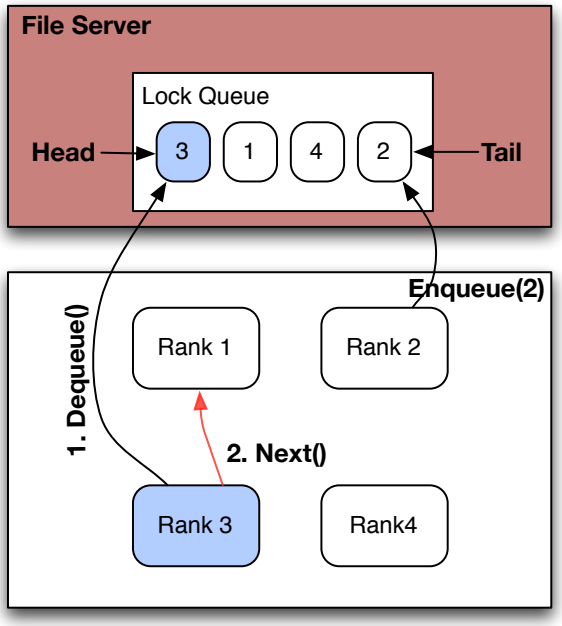

Figure 2: Diagram of the lock queue algorithm.

A slightly modified version of this algorithm was implemented in [18], using MPI point-to-point messages to wake sleeping clients and MPI one-sided operations to implement the shared queue. However, due to restrictions of the MPI one sided operations, it is not feasible to implement a true shared queue. Instead, a shared array was used, recording who has the lock and who is waiting for the lock. In this array, each client has its own dedicated slot. Upon releasing the lock, the current owner of the lock examines the array (using a one-sided *get* operation) to see if any other client is sleeping for the lock. If so, it selects a client and wakes it using a point-to-point message.

There are a number of drawbacks associated with this approach. For one, MPI does not guarantuee that one-sided operations will complete without active support from the remote party (i.e. the node containing the accessed memory location). This means that, even though the lock is free, it cannot be obtained without cooperation from unrelated clients. If those clients do not make MPI calls, for example because they are computing, the lock cannot be obtained. Second, approximating the queue with a shared array, while starvation free, does not guarantee fairness. Also, since each client that needs to obtain the lock requires a dedicated entry in the array, all clients must be known before creating the lock and need to agree on a unique integer based ordering.

Using atomic operators to implement a shared

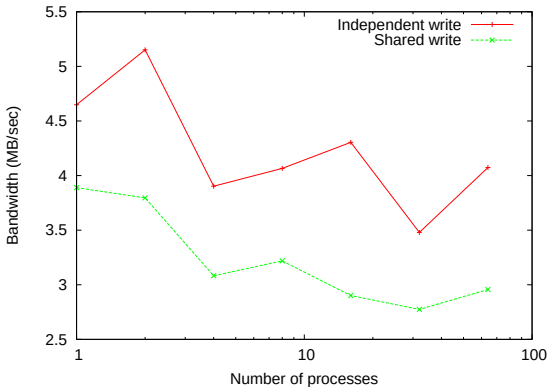

Figure 3: Independent vs. Shared file pointer writes.

Figure 4: Independent vs. Shared file pointer reads.

queue, we are able to avoid these drawbacks. Clients only need to agree on the file and attribute name to gain access to the queue, making it relatively easy to implement file locking between unrelated programs. Since the file server already queues requests and processes attribute operations atomically, true fairness is guaranteed. And because the file server is always ready to service file operations, progress is ensured without requiring other clients to call the library.

5. Experimental Results

We implemented the atomic operators as extended attributes in the PVFS file server. This allowed us to use unmodified client software to perform atomic operations on files. Using these atomic operators, we implemented the shared file pointer interfaces in ROMIO, and implemented the distributed locks using `getxattr` system calls and MPI for communication between processes. Tests were run on the breadboard cluster, a 40 node x86_64 commodity cluster at Argonne National Laboratory.

5.1. Shared File Pointers

We ran a micro-benchmark comparing performance when writing a very small amount of data (4k per process) to a file on PVFS using both shared file pointer methods and explicit access routines. For the explicit case (using the MPI write_at and read_at routines), we precomputed offsets to ensure no overlapping writes. The shared case relies on correct updating of the shared file pointer.

As can be seen in Figures 3 and 4, the shared file pointer routines have somewhat more overhead than the explicit access approach. We attribute this difference to the need for the shared file pointer routines to perform an extra metadata access. Because each process does such a small amount of I/O, additional metadata calls appear to have an outsized effect.

5.2. Distributed Locks

Using atomic operators and the distributed locking algorithm explained in 4.2, advisory file locking can be implemented even on filesystems such as PVFS which do not natively support `fcntl()`. Once advisory file locks are available, they subsequently be used to implement MPI-IO atomic mode.

To evaluate the performance of our lock implementation, we compared its performance against the pure MPI implementation (described in [18]) and the more traditional approach of using lockfiles. Tests were executed on breadboard, a linux cluster at Argonne National Laboratory. The MPI library was mpich version 1.1 using TCP over gigabit ethernet as the communication network.

Method	Acquiring	Releasing
Active Attributes	3.5ms	3.5ms
MPI	0.22ms	0.22ms

Table 1: Lock performance in the absence of contention.

As a first test, we measured the time to obtain and release a lock. When using atomic operators releasing a lock only depends on the process releasing the lock and the file server, so the time needed to release the lock is independent of the other clients. For MPI one-sided implementation, this is not true. To obtain results independent of the specific calling sequence of other clients, we dedicated a client to managing the lock. This means the times listed in table 1 represent a lower bound for the performance of the MPI method. The timings given are valid when there is no contention for the lock.

In a second test, we simulated a workload consisting of a number of independent, atomic updates to a shared file. Each available client is responsible for a subset of the updates, meaning the aggregated number of updates among all the clients is always the same. In total, 200

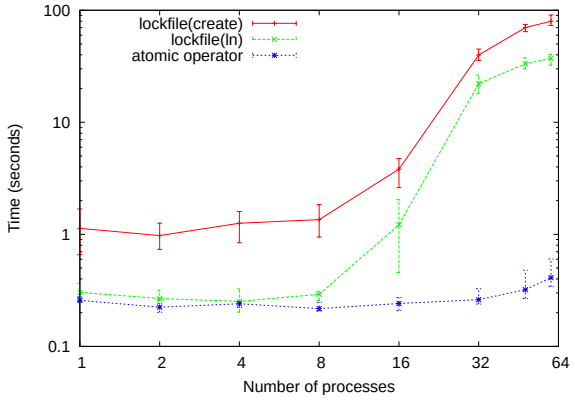

Figure 5: Time needed to perform a set of atomic updates.

updates are made to the file, where each update writes 4Kb of data.

Since, when using the one-sided implementation, the lock cannot be obtained while the client is busy performing I/O, we compared our atomic operator implementation to the more common practice of using file locks. On most filesystems, as filenames have to be unique, creating a file is an atomic operation. Therefore, the ability to create a certain file with a specific, previously agreed on name, can serve as an atomic operation similar to obtaining a lock. When the owner of the lock is finished, the file is removed, giving other clients a chance to create the file and obtain the lock. If a client is unable to create the lockfile (because it already exists), it has to try again until it succeeds, increasing the load on the file server for each waiting client.

Unfortunately, especially on network filesystems, creating a file is not always atomic. There are a number of workarounds, but it is clear that using a lockfile is not an ideal solution. We implemented two lockfile methods, a straightforward implementation using only lockfiles which might fail on certain network filesystems, and a more complicated implementation relying on hard links to achieve better portability. The results are shown in Figure 5. The effect of the load generated by blocked clients in the lockfile methods is clearly visible as the number of clients increases. The queue implementation, which relies on the queue to record who is next in line to receive the lock, does not have this problem and subsequently scales much better even if the number of clients increases.

6. Conclusions and Future Directions

Parallel applications and I/O libraries routinely need to coordinate access. While current parallel file systems provide some basic facilities for coordination (e.g., fcntl() locks, POSIX atomicity), these facilities are heavy weight; they do not serve as good building blocks for more complex coordination activities such as shared file pointers. Further, facilities such as file system locking used to enforce POSIX atomicity introduce shared state into the system, placing the burden of failure recovery on the file system.

In this paper we introduce the use of atomic operations on extended attributes as an alternative method for coordination using the parallel file system. We integrate these atomic operations into the parallel file system PVFS, by providing support for atomic operations on files through extended attributes. Finally, We discuss two common distributed application patterns that can leverage the coordination primitives we provide, and show experimental results using those primitives.

Using extended attributes allows us to leverage existing file system interfaces and operating system support, but has drawbacks that must be addressed. More research is needed to identify a complete set of atomic primitives that allow a wide variety of applications to perform coordination within the file system. Better understanding of what primitives are needed will naturally lead to more appropriate file system interfaces.

Acknowledgments

The authors would like to thank the PVFS community for their efforts in making PVFS a successful part of HPC storage systems. This work was supported by 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.

References

[1] W. Snaman Jr and D. Thiel, "The VAX/VMS distributed lock manager," *Digital Technical Journal*, vol. 5, pp. 29–44, 1987.

[2] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *In Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, 2002.

[3] W. Liao and A. Choudhary, "Dynamically adapting file domain partitioning methods for collective I/O based on underlying parallel file system locking protocols," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press Piscataway, NJ, USA, 2008.

[4] K. Preslan, S. Soltis, C. Sabol, M. O'Keefe, G. Houlder, and J. Coomes, "Device locks: Mutual exclusion for storage area networks," in *Proceedings of the Seventh NASA Goddard Conference on Mass Storage Systems and Technologies*, 1999.

[5] *SCSI Device Memory Export protocol. Version 0.9.8.* [Online]. Available: http://www.t10.org

[6] K. W. Preslan, A. Barry, J. Brassow, R. Cattelan, A. Manthei, E. Nygaard, S. V. Oort, D. Teigland, M. Tilstra, M. OKeefe, G. Erickson, and M. Agarwal, "Implementing journaling in a linux shared disk file system," in *Proceedings of the Eighth NASA Goddard Conference on Mass Storage Systems and Technologies*, 2000.

[7] K. W. Preslan, A. Barry, J. Brassow, M. Declerck, A. J. Lewis, A. Manthei, B. Marzinski, E. Nygaard, S. Van Oort, D. Teigland, M. Tilstra, S. Whitehouse, and M. O'Keefe, "Scalability and failure recovery in a linux cluster file system," in *ALS'00: Proceedings of the 4th annual Linux Showcase & Conference*. Berkeley, CA, USA: USENIX Association, 2000, pp. 10–10.

[8] *Data sharing with a GFS storage cluster*. [Online]. Available: http://www.redhat.com/magazine/006apr05/features/gfs/

[9] A. Ermolinskiy, D. Moon, B.-G. Chun, and S. Shenker, "Minuet: rethinking concurrency control in storage area networks," in *FAST '09: Proccedings of the 7th conference on File and storage technologies*. Berkeley, CA, USA: USENIX Association, 2009, pp. 311–324.

[10] R. Ross, D. Nurmi, A. Cheng, and M. Zingale, "A case study in application I/O on Linux clusters," in *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*. ACM New York, NY, USA, 2001, pp. 11–11.

[11] *The Linux open(2) manpage*, section O_EXCL. [Online]. Available: http://www.kernel.org/doc/man-pages/online/pages/man2/open.2.html

[12] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *OSDI '04: Proceedings of the 5th symposium on Operating Systems Design and Implementation*, 2004.

[13] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," *SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 29–43, 2003.

[14] P. F. Corbett and D. G. Feitelson, "Design and implementation of the Vesta parallel file system," in *Proceedings of the Scalable High-Performance Computing Conference*, 1994, pp. 63–70.

[15] *Paragon System User's Guide*, Intel Supercomputing Division, 1993.

[16] P. Pierce, "A concurrent file system for a highly parallel mass storage system," in *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*. Monterey, CA: Golden Gate Enterprises, Los Altos, CA, March 1989, pp. 155–160.

[17] C. S. Freedman, J. Burger, and D. J. Dewitt, "SPIFFI — a scalable parallel file system for the Intel Paragon," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 11, pp. 1185–1200, November 1996. [Online]. Available: http://www.computer.org/tpds/td1996/l1185abs.htm

[18] R. Latham, R. Ross, and R. Thakur, "Implementing mpi-io atomic mode and shared file pointers using mpi one-sided communication," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 2, pp. 132–143, 2007.

[19] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective I/O in ROMIO," in *Frontiers of Massively Parallel Computation, 1999. Frontiers' 99. The Seventh Symposium on the*, 1999, pp. 182–189.

[20] R. Thakur, E. Lusk, and W. Gropp, "Users guide for ROMIO: A high-performance, portable MPI-IO implementation," vol. 234, 2000.

[21] A. Ching, R. Ross, W. keng Liao, L. Ward, and A. Choudhary, "Noncontiguous locking techniques for parallel file systems," in *Proceedings of Supercomputing*, November 2007.

[22] "Lustre file system," http://www.sun.com/software/products/lustre/.

[23] J. Piernas, J. Nieplocha, and E. Felix, "Evaluation of active storage strategies for the lustre parallel file system," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. ACM New York, NY, USA, 2007.

[24] A. Devulapalli, D. Dalessandro, and P. Wyckoff, "Data structure consistency using atomic operations in storage devices," in *SNAPI '08: Proceedings of the 2008 Fifth IEEE International Workshop on Storage Network Architecture and Parallel I/Os*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 65–73.