MPI-IO/L: Efficient Remote I/O for MPI-IO via Logistical Networking

Jonghyun Lee*, Robert Ross*, Scott Atchley†, Micah Beck‡, Rajeev Thakur*

*Argonne National Laboratory
Math. and Comp. Sci. Div.
Argonne, IL 60439 USA
{jlee, rross, thakur}@mcs.anl.gov

†Myricom, Inc.
Oak Ridge, TN 37830 USA
atchley@myri.com

‡University of Tennessee
Dept. of Comp. Sci.
Knoxville, TN 37996 USA
mbeck@cs.utk.edu

Abstract

Scientific applications often need to access remotely located files, but many remote I/O systems lack standard APIs that allow efficient and direct access from application codes. This work presents MPI-IO/L, a remote I/O facility for MPI-IO using Logistical Networking. This combination not only provides high-performance and direct remote I/O using the standard parallel I/O interface but also offers convenient management and sharing of remote files. We show the performance trade-offs with various remote I/O approaches implemented in the system, which can help scientists identify preferable I/O options for their own applications. We also discuss how Logistical Networking could be improved to work better with parallel I/O systems such as ROMIO.

1 Introduction

As networking technologies improve, scientists are running an increasing number of applications in distributed environments, with the frequent need to store and retrieve data at remote locations [8]. Traditionally, scientists have performed remote I/O by temporarily copying, or *staging*, remote files in their entirety to local disks. Although staging seeks to boost the I/O performance by colocating data with the application, it imposes several problems. For example, it does not allow applications to directly access remote files, requiring extra disk I/O. It can also cause consistency problems and excessive data transfer for partial file access. Moreover, staging is often done manually and is thus not convenient.

A better remote I/O system should address the following I/O needs typically required by many scientists.
Functionality. Direct access to any portion of remote files through a convenient interface should be possible. Since many scientific codes are parallel, supporting parallel I/O is highly beneficial.
Performance. Remote I/O is often slow because of low wide-area network bandwidth and the amount of data to be accessed. Thus, it is important to reduce apparent remote I/O cost [2, 10, 13].
Management. Scientific data files can be replicated or striped across multiple storages for fault tolerance or faster access. Efficient management of the information about each replica or stripe (e.g., its physical location and mapping to the logical file) is helpful, especially with a number of logical and physical files.
Sharing. Scientific data is often shared among a group of people. Distributing information about files and accessing the files using such information should be easy.

This work addresses the above issues by coupling a parallel I/O library with remote I/O functionality. We chose the ROMIO [15] implementation of MPI-IO [12] for the testbed I/O library and Logistical Networking [5] for the remote I/O and file management component. MPI-IO is the de facto parallel I/O interface standard, used both directly by applications and by high-level libraries [1, 11]. Supporting remote I/O via MPI-IO enables many applications to perform remote I/O transparently without code changes. Logistical Networking provides powerful remote I/O mechanisms, including efficient transfer by concurrent data streams and intelligent download schemes [13]. It also flexibly describes the relationship between a logical file and its associated physical files with a portable XML file, thereby easing the sharing.

The contributions of this work are as follows. First, we identify the design issues for a Logistical Networking-based remote I/O facility for a parallel I/O system and provide an implementation with ROMIO, called MPI-IO/L. Second, we optimize basic I/O operations for MPI-IO/L and discuss the trade-offs of the proposed approaches, helping users select the preferable remote I/O options for their applications. Third, we identify ways to make Logistical Networking work better with parallel I/O systems such as ROMIO.

This work was supported in part by the U.S. Dept. of Energy under Contract W-31-109-ENG-38. An extended version of this manuscript is available as ANL/MCS-TM-290.

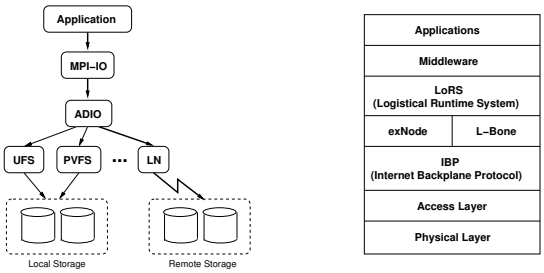

Figure 1. (a) ROMIO's layered architecture; (b) Network Storage Stack.

1.1 MPI-IO and ROMIO

MPI-IO provides the standard interface for parallel I/O from multiple processes to a common file on a shared file system. Each process can access the file either independently or cooperatively with other processes (collective I/O[1]). MPI-IO defines consistency semantics, which state what happens when multiple processes concurrently access the same file. The default semantics in MPI-IO guarantee that the changes written by a process will be immediately visible to that process, but not to the other processes until explicit synchronization is performed. When multiple writes are performed on the same region, the result is undefined. This approach is more relaxed and more suitable for parallel I/O than is the strict sequential consistency required by POSIX I/O, and it provides opportunities to optimize the performance within the MPI-IO implementation.[2]

ROMIO is a high-performance and portable implementation of MPI-IO. For high performance, ROMIO optimizes noncontiguous I/O operations through data sieving and collective I/O through two-phase I/O [15]. ROMIO achieves portability through an internal I/O layer called Abstract Device I/O (ADIO) [15]. ADIO defines a set of basic I/O interfaces that are used to implement more complex, higher-level I/O interfaces such as MPI-IO. For each supported file system, ADIO requires a separate implementation ("module") of its I/O interfaces. Figure 1(a) depicts this architecture.

1.2 Logistical Networking

Logistical Networking provides scalable and sharable storage resources and services for distributed applications. The center of this technology is the Network Storage Stack (Figure 1(b)), which was modeled after the Internet Protocol stack. The bottom layers (physical and access) consist of storage media such as disk and memory and storage services such as the disk driver. The middle layers are the core of Logistical Networking; each is described below. The top layers include the middleware layer, which provides services using underlying components such as this work, and the application layer.

[1]Collective I/O optimizes parallel I/O operations by using the global knowledge of data distributions in memory and file [15].

[2]To force the sequential consistency within MPI-IO, a special "atomic" mode can be used, or writes can be temporally separated from each other by surrounding each with sync operations.

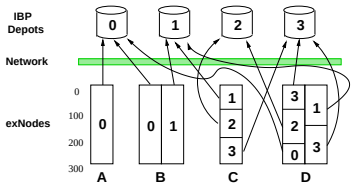

Figure 2. Sample exNodes.

IBP (Internet Backplane Protocol). IBP allows anyone to share the storage space on her machine over the network. Unlike FTP servers, these IBP *depots* do not maintain user accounts, file system hierarchies, and the like. A client can request an allocation (a specific amount of space) for a specific amount of time from any depots and can access and share the allocation until it expires. Each depot is preset with the total amount of storage space that it is willing to share and the maximum time allowed for any single allocation.

IBP implements only six storage operations. Among them, the operation `allocate()` either grants or denies client requests based on the depot's space availability and the duration policy. For a granted request, the depot sends three capabilities (keys) to the client, needed for write, read, and manage (change the allocation properties such as size) access to the allocation. A client can share an allocation with others by giving them a subset of capabilities (e.g., read only) associated with it to control their access to the allocation. The operations `store()` and `load()` write to and read from an allocation, respectively. While read can begin from any offset within the allocation, writes are currently append only.

L-Bone (Logistical Backbone). L-Bone is an LDAP-based server that catalogs and polls the available public depots periodically to determine available storage. Clients can send a request to an L-bone server for a list of depots that meet specific storage criteria.

ExNode. When a logical file is striped or replicated across depots, multiple allocations and capabilities are associated with the file. The exNode is a data structure that aggregates allocations and provides a mapping from the logical view of a file to the actual allocations, analogous to Unix inodes. Unlike the inode, however, the exNode allows for varying size allocations, data replication, and arbitrary metadata for both the global exNode and individual mappings. The exNode can be *serialized* to XML for sharing between processes or clients.

Figure 2 shows the exNodes for four logical files. The numbers shown in each exNode indicate which depot stores an allocation for a certain portion of each file. Here, A is stored in an allocation on depot 0, while B is replicated on depots 0 and 1. C is striped over three

```
<exnode:mapping>
  <exnode:metadata name="alloc_length" type="integer">100</exnode:metadata>
  <exnode:metadata name="alloc_offset" type="integer">0</exnode:metadata>
  <exnode:metadata name="exnode_offset" type="integer">100</exnode:metadata>
  <exnode:metadata name="logical_length" type="integer">100</exnode:metadata>
  <exnode:read>ibp://depot2:[port]/[key string]/READ</exnode:read>
  <exnode:write>ibp://depot2:[port]/[key string]/WRITE</exnode:write>
  <exnode:manage>ibp://depot2:[port]/[key string]/MANAGE</exnode:manage>
</exnode:mapping>
```

Figure 3. A serialized mapping in XML.

depots, and D is both replicated and striped. Figure 3 shows the serialized exNode in XML for a mapping to an allocation of C stored in depot 2.

LoRS (Logistical Runtime System). LoRS provides command line tools and APIs that automate the finding of depots via the L-Bone, creating and using allocations and capabilities, and creating exNodes. The remote I/O facility described here was built by using the LoRS APIs.

2 Design

As mentioned, MPI-IO/L was implemented by adding a Logistical Networking-specific ADIO module to ROMIO. MPI-IO/L maintains the file *metadata* (e.g., the mapping between logical and physical file contents), contained in the exNode, locally where the application runs, while the file data can be stored remotely. The MPI-IO consistency semantics state that each file data and metadata update need not be immediately visible to other processes unless explicit synchronization methods are used. Accordingly, in MPI-IO/L each process keeps a separate exNode in its memory and only locally updates it for each write. At the user's request (by calling sync or close), the exNodes are synchronized among processes, and thus the changes made by other processes become visible. This coarse-grained synchronization can effectively reduce the amount of communication.

Based on this file access model, we designed and implemented the functions for the ADIO module as follows.[3]

Open and Close. `LN_Open` and `LN_Close` collectively open and close a remote file from multiple processes. When opening a file, the name of its XML exNode file is passed to `LN_Open` along with other attributes such as file access mode. MPI-IO/L assumes that the exNode file will be accessed from a local file system and has only the rank 0 (root) process read and write the file.

When called, `LN_Open` creates two data structures in each process's memory. First, it creates an exNode that contains allocation- and mapping-related information for the remote file. If the given XML file exists, the root first reads it and broadcasts its content to the other processes. Then, each process *deserializes* the XML file content into an empty exNode. If the file does not exist, each process creates only an empty exNode.

Next, `LN_Open` creates a *depotpool* on each process. This is a list of depots that will be used for data storage and retrieval. If a file is opened for read-only, the depotpool is created by extracting from the XML file all the depots that provided allocations for previous writes. If the file is opened for write-only, an L-Bone server is contacted to find depots that satisfy the user's storage requirements.These requirements are passed to `LN_Open` as hints by using an `MPI_Info` object. The user can also provide a list of known depots that she wants to use. LoRS provides separate APIs for depotpool creation for read-only and write-only cases.

When a file is opened for both read and write, both depot extraction and search should be performed. However, LoRS currently does not provide a single API for both operations. Moreover, it does not allow the two APIs used for read-only and write-only cases to be combined. MPI-IO/L deals with this problem by manually extracting depots from the XML file and adding them to the L-Bone search results. This is not a desired solution, however, because LoRS maintains only one depotpool, and thus the depots extracted for future reads are included in the same depotpool that contains the ones selected for writes. Hence, the read depots might be used for future write operations, and if they fail to meet the user's storage requirements, an error will occur.

Other hints can be passed at file open. They include the hostname and port number of a preferred L-Bone server, the size of the unit at data transfer (called *block*), the number of replicas to create at each write, the number of threads to be used for concurrent transfers, the size of the memory buffer for buffered I/O (described later), and options for MPI-IO/L specific optimizations. After creating the two data structures on each process, `LN_Open` initializes internal variables according to the passed hint values, allocates buffer memory if buffered I/O is enabled, and then returns.

`LN_Close` calls `LN_Sync` to synchronize exNodes from all the processes and serialize the synchronized exNode to the XML file, frees internal data structures, and returns. `LN_Sync` is described in detail later.

Contiguous I/O. MPI-IO/L provides two modes of contiguous I/O. *Direct I/O* issues remote I/O requests immediately. *Buffered I/O* temporarily buffers the write data using preallocated memory. The buffer holds one contiguous region of the file, whose coverage can change dynamically. If the write data cannot be buffered in the current buffer in its entirety or will not form a contiguous extent when combined with previously buffered data, then the buffer is flushed by a direct write before buffering the new data. If the write data is larger than the size of the allocated buffer, only a buffer size of data at the end is buffered; the rest is written directly. Buffered reads are carried out from the buffer if it contains at least a portion of the requested data, and the nonbuffered portion is read directly.

Buffered I/O is not asynchronous because LoRS does not have an asynchronous I/O interface yet. Rather, it

[3]The implementation is partly based on libxio version 0.2, which provides standard Unix I/O interfaces using Logistical Networking.

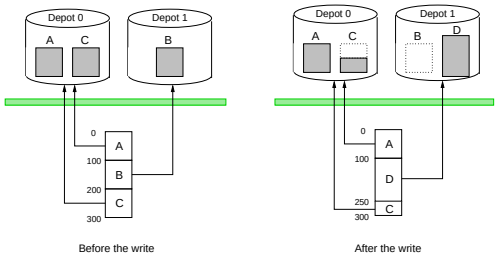

Figure 4. Status of an exNode before and after "Write 150 bytes at offset 100."

is a write optimization to coalesce several small writes into few large ones to reduce both per-operation overhead and the number of allocations and mappings created.[4] It can also reduce the number of actual I/O operations to be performed if a certain buffered region is overwritten or read multiple times. If asynchronous I/O becomes available in LoRS, buffered I/O can further improve I/O performance through write-behind and prefetching.

`LN_WriteContig` first checks the current I/O mode. Direct write is chosen when the buffer size is set to 0. Otherwise, a buffered write is performed, which in turn can issue direct writes. Direct write is performed by the LoRS function `lorsSetStore()`, which creates a new allocation, stores the data, creates mappings to it, and places the mapping into a set.[5] Data larger than the block size is further divided to issue smaller writes.

A write to an unwritten portion of a file is performed simply by calling `lorsSetStore()` and adding the newly created mapping(s) to the exNode. However, an overwrite of existing mappings must go through four steps. First, mappings that overlap with the write request are removed from the exNode and put into a set. Second, each mapping in the set is either removed, if the logical extent that it points to will be completely overwritten, or trimmed if it will be only partially overwritten. Third, `lorsSetStore()` is called to write the requested data and create new mappings to new allocations in another set. Fourth, the two sets are merged, and the mappings in the merged set are added to the exNode. Figure 4 shows how an exNode is changed after an overwrite. The original exNode contains three mappings to three allocations (A, B, and C) across two depots for a 300-byte-long logical file. When an operation that writes 150 bytes at offset 100 is issued, the mappings to B and C overlap with the write request, and they are either removed or trimmed accordingly. The data is written to to a new allocation D and the exNode is updated. In the figure, only the shadowed portion of each allocation is mapped to a portion of the logical file. For example, after the write, only the second half of C is mapped to the file. B is not mapped to the file anymore, but the allocation remains until it expires or is revoked explicitly. If the depots supported write-at-offset, the overwrite could be performed without additional allocation and the change of mappings in the exNode.

`LN_ReadContig` also checks the I/O mode first. If direct read is set, mappings that overlap with the requested extent are identified and put into a set. If the mappings in the set form a contiguous extent, `lorsSetLoad()` is called once for the set, which reads the allocations pointed to by the mappings in the set. If holes[6] are formed by unwritten portions in the extent, however, the current implementation of `lorsSetLoad()` will return an error when attempting to read that extent. To get around this problem, `LN_ReadContig` first identifies each contiguous region in the whole extent and which mappings will form the region. Then, a separate `lorsSetLoad()` for each contiguous region is issued, and the results are combined in the user buffer.

Noncontiguous I/O. Noncontiguous I/O can be performed in a few ways. A naïve approach issues a separate I/O request for each contiguous region contained in the extent, but this will incur overhead for each I/O operation issued. Another option, implemented by ROMIO, is data sieving [15]. For a noncontiguous read, data sieving reads the whole extent of the I/O request and picks out only the regions that are requested. For a noncontiguous write, data sieving reads the whole extent of the request into a buffer, modifies the requested regions in the buffer, and writes the whole buffer back. Since the whole procedure must be performed atomically, however, data sieving write works only with file systems that provide file locking, making it unsuitable for MPI-IO/L.

`LN_WriteNoncontig` provides two different options. It can simply reuse ROMIO's naïve noncontiguous write routine. Alternatively, it can optimize noncontiguous writes by exploiting the fact that multiple mappings can be flexibly generated to a single allocation. The data to be written to the noncontiguous extent is first packed into a contiguous memory buffer. Then, the packed data buffer is written into one or more allocations (depending on the packed data size relative to the block size). Next, a separate mapping is created between each contiguous region in the original extent and its corresponding portion in the newly created allocation in the exNode. This method resembles a log-structured file system in that it stores noncontiguous regions contiguously. However, LoRS currently does not provide a single API that creates multiple arbitrary mappings to a single allocation (`lorsSetStore()` creates only one mapping per allocation). MPI-IO/L gets around this problem by calling `lorsSetStore()` once to store the packed data, memory copying the newly created mapping(s) multiple times,

[4] For each write request, LoRS currently creates a separate allocation and a mapping to it.

[5] In LoRS, a set is defined as a collection of mappings, on which a common operation can be executed in batch.

[6] We distinguish two types of holes. If the holes result from unwritten portions of the file, we treat them as the regions filled with unknown values. But, if holes are formed by already expired allocations, reading such region is considered to be an error.

modifying the offset and length fields for both logical and physical allocation in each copied mapping accordingly, and adding the modified mappings, not the ones created by `lorsSetStore()`, to the exNode. All these procedures are performed in memory.

The second approach is expected to perform better than the naïve approach because it can significantly reduce the number of write calls and thus the overhead associated with each invocation. Moreover, the manipulation of mappings is done in memory and should be efficient. However, it will create the same number of mappings as the naïve approach. Also, even though a set of contiguous regions is packed and stored in a single allocation, future reads whose extents overlap with the noncontiguous write extent cannot recognize this fact, and thus be optimized accordingly, because the exNode currently does not provide an efficient way to describe such information. Users can choose which write option to use by providing a hint at file open. The default is the optimized approach.

`LN_ReadNoncontig` also provides two options. The naïve approach reads each contiguous region in the extent with a separate LoRS call. Alternatively, data sieving can be used, because it does not require locking for reads. Both approaches have trade-offs. While the naïve read involves many more read calls and thus is likely to incur higher overhead, data sieving causes extra data access, which could be expensive for remote file systems. Careful performance study is needed to decide which to choose for a given noncontiguous read request. The user can choose the method of noncontiguous reads by providing a hint at file open. The default is data sieving.

Since noncontiguous I/O implementations eventually call file system-specific contiguous I/O calls, buffered I/O can still be used, although each buffer is supposed to hold a contiguous extent and thus is not of much help for sparse noncontiguous I/O. The only exception is the optimized noncontiguous write, which does not call contiguous write routine. In this case, if the current buffer extent overlaps with the extent of the noncontiguous write request, we simply flush the buffer before taking actions for the noncontiguous write.

Sync. An MPI-IO sync operation should cause all previous writes to be transferred to the storage device so that the changes (both to file data and to metadata) made by one process will be visible from other processes. However, IBP currently does not provide a mechanism for a file data sync. Thus, `LN_Sync` synchronizes exNodes among processes only by having each process broadcast the mappings that it has modified since the last synchronizaion point and flushes the buffer on each process if buffered I/O is used.

In addition to the in-memory exNode synchronization, `LN_Sync` combines the current content of the XML file with the in-memory exNodes to incorporate changes made by other groups of processes that concurrently opened the same file. If the other processes wrote some data to the file and called `LN_Sync`, the XML files contain new mappings that should be visible. After these steps, every process will have identical in-memory exNode, and then the root will serialize it to the XML file.

Other Functions. `LN_Delete` simply deletes the locally stored XML file. The allocations contained in the file will be revoked when their expiration occurs. A more proactive approach would be freeing the allocations at the time of delete, but this could cause problems if some of the allocations were shared with other exNodes.

`LN_Resize` is implemented in two ways. First, if a file needs to be expanded, the root writes one byte of null data at the last offset. Second, if a file needs to be shrunk, the mappings in the exNode are trimmed on each process. `LN_Prealloc` has the root write 0s to the file from the next byte of the current last offset up to the desired size. Both functions are collective and require exNode synchronization so that non-root processes can obtain the correct file size after the calls.

Atomic mode and shared file pointers have not been implemented yet. The current implementation of both atomic mode and shared file pointers in ROMIO requires global locking at either the file system or the MPI level. For MPI-IO/L, it has to be at the MPI level because IBP does not provide locking. An implementation of byte-range locks using MPI-2 passive target RMA (Remote Memory Access) has been proposed [16]. However, because of the limitation in the current version of MPICH2 that requires MPI functions to be called for progress at the target, we decided to wait until this deficiency is addressed.

3 Results

We conducted experiments between the Jazz Linux cluster at Argonne National Laboratory and a selected depot at the University of Tennessee at Knoxville. Jazz comprises 350 nodes, each equipped with a 2.4 GHz Pentium Xeon, either 1 or 2 GB of RAM, and 80 GB of local disk space. Jazz nodes are connected by both Myrinet 2000 and Fast Ethernet, and we used Fast Ethernet for the experiments to emphasize that remote I/O performance is dominated by wide-area network bandwidth rather than by the local interconnect. The XML files were created and accessed on Parallel Virtual File System (PVFS). All the numbers were averaged over five or more runs; the error bars show a 95% confidence interval.

3.1 Synthetic Benchmark Performance

We devised a synthetic benchmark that performs both contiguous and noncontiguous I/O with various parameters from a single Jazz node to a single depot.

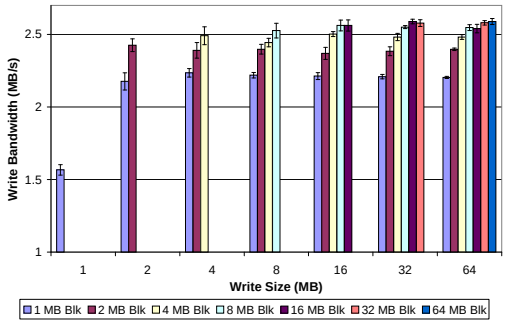

Figure 5. Single-threaded contiguous writes.

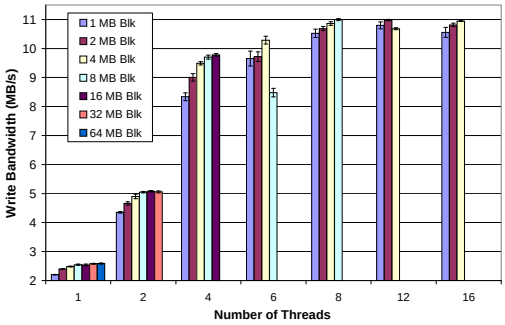

Figure 6. Multithreaded contiguous writes.

Contiguous I/O. Figure 5 shows the direct remote write bandwidth for up to 64 MB of data, with a single thread.[7] For each transfer, we varied the block size, to observe the effect of block size on performance. The graph shows that the write bandwidth increases up to certain point (2 MB here) but remains more or less the same after that, because per-I/O request overhead becomes negligible compared to the actual transfer cost as the data size increases. Also, for each write size, larger block sizes increase write bandwidth because, with larger block size, the number of I/O calls issued to transfer the same amount of data decreases, and hence the aggregate per-I/O request overhead also decreases. Because of the space constraint, we show only the write performance here. The reads showed a similar trend.

We also compared the performance of directly calling `lorsSetStore()` and `lorsSetLoad()` to the numbers shown above, to measure the MPI-IO and ADIO layer overhead. The overheads observed were all less than 1% of LoRS function costs, and thus negligible.

Figure 6 shows the multithreaded write performance for 64 MB of data. We used up to 16 threads for concurrent transfer and also varied the block size. As mentioned, the unit of transfer in LoRS is a block, and each thread transfers a block of data at a time. For each block size, the number of threads times the block size did not exceed the amount of data to be accessed.

The graph shows that the bandwidth increases as the number of threads increases up to a certain point (about 8 here) and remains more or less constant after that. It also shows that with the same number of threads, larger blocks perform better. As the block size increases, however, the maximum degree of concurrency achievable decreases, and thus the overall maximum write bandwidth cannot be reached with larger blocks. There are some exceptions where larger blocks perform worse than smaller blocks with the same number of threads, such as 8 MB block with 6 threads, because the amount of data to be written cannot be a product of any of the block sizes used in the experiments and these numbers of threads and thus only a subset of threads will transfer the data at the last round of transfer. This situation suggests that the number of threads and block size should be carefully chosen according to the expected amount of data to access when large transfers are common. The threaded read performance showed similar trends to the writes.

[7] As mentioned, LoRS provides multithreaded data transfer using pthreads.

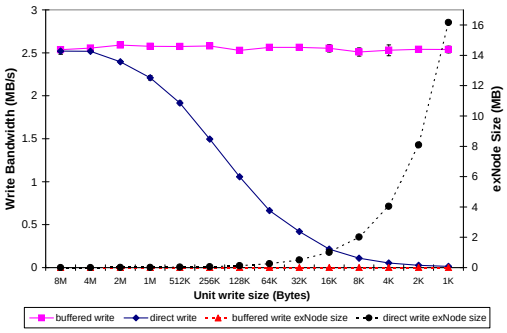

Figure 7. Buffered write vs. direct write.

Figure 7 compares the performance of buffered writes to that of direct writes. A total of 16 MB of data was written using different write sizes, with a 16 MB block size, a single thread, and a 16 MB buffer for buffered writes. Direct write performs the same number of write operations as the number of write operations issued, while buffered write coalesces all the writes and performs the write only once. The buffered write performance shown in the graph includes the sync cost, while direct write does not. The result shows that the direct write performance drops significantly with larger number of write operations because of per-operation overhead. On the other hand, the buffered write performance stays high regardless of the write size, because only one write call is issued in all cases. In our extreme case with 1 KB write size, buffered write performs more than 189 times better than direct write. This suggests that buffered write should be considered when a series of sequential and contiguous writes are expected and extra memory is available.

The dotted lines in the graph show the size of the XML file that each configuration creates. For direct writes, the size of the XML file increases as the write size decreases and thus the number of created mappings increases. With 1 KB write size, the XML file is larger than 16 MB. On the other hand, buffered I/O minimizes

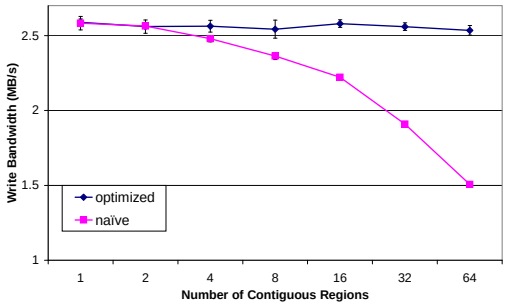

Figure 8. Noncontiguous write performance.

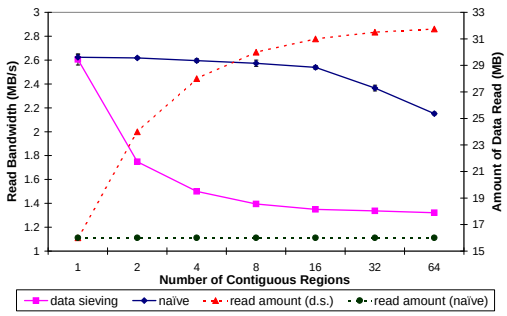

Figure 9. Noncontiguous read performance.

the number of writes performed to one, and the resulting XML file is a little over 1 KB. The size of XML files affects the performance of file open, close, and sync that read and write the XML file. The graph does not show the effect of sync in direct I/O because the sync cost is quite small compared to the actual write cost, but the sync-only cost for direct write with 1 KB write size was longer than the buffered write and sync cost combined.

Noncontiguous I/O. Figure 8 shows the performance of the two noncontiguous write approaches with a 16 MB block size and a single thread. We fixed the total amount of data to be written to 16 MB but varied the number of contiguous regions in the extent up to 64. In this access pattern, the stride is twice the size of each contiguous region; that is, each contiguous region is followed by a hole of the same size. The graph shows that the optimized write performance is almost constant regardless of access pattern, because it issues only a small number of write calls (only one here) and the in-memory mapping manipulation is fast. However, the naïve write performance degrades as the number of regions increases, because of multiple I/O call overhead. The performance gap is expected to increase when finer access patterns are used. The size of the resulting XML file is almost the same for both approaches because the optimized approach does not reduce the number of mappings, only the number of allocations.

Figure 9 presents the noncontiguous read performance using the same configuration. Here, 32 MB of data was contiguously written to the depot prior to the access. The graph shows that both data sieving and naïve read performances decrease as the number of contiguous regions increases, but for different reasons. The naïve read

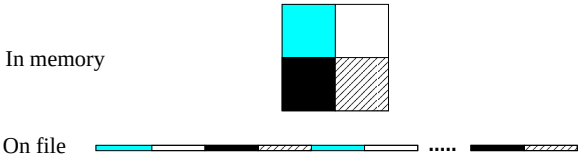

Figure 10. A 2×2 tile mesh and its file representation.

performance degradation is due to the multiple I/O call overhead. On the other hand, the decreased performance for data sieving is due to the fact that the extent of the noncontiguous read (and the amount of unnecessary data read) increases with the number of contiguous regions. The dotted lines show the total amount of data that should be read for each approach.

As we keep increasing the number of regions, the data sieving performance is expected to converge to half of the 16 MB contiguous read performance, as the total extent to be read converges to 32 MB. However, the naïve approach performance is likely to keep decreasing as we increase the number of regions, because of the excessive number of read operations issued, and it is expected to start performing worse than data sieving after some point. Thus, it is important to find the balance point and choose the faster approach according to the access patterns.

We also tested noncontiguous read performance where the data is read from a noncontiguously written file. In this case, however, data sieving loses its benefit for finer accesses because now a separate mapping exists for each contiguous region and the data sieving will have to issue multiple reads, one for each mapping, even though it reads a contiguous extent from the logical file.

3.2 Tiled I/O Performance

The tiled I/O benchmark measures the performance of representative I/O access patterns common in many parallel scientific applications—parallel I/O for distributed multidimensional arrays. Tiles (subarrays) are created by dividing a 2-D data set along each dimension, with each process accessing a distinct tile. The tiles are written to and read from a global row-major order file (Figure 10). In our experiments, 4096×4096 arrays with 4-byte elements were distributed across a 4×4 tile mesh, totaling 64 MB of data per array.

We tested both collective and noncollective approaches. Collective I/O may reorder data among processes to issue fewer, larger contiguous I/O requests. For example, for each global array access with the tiled I/O, the data is reorganized so that ith process can contiguously access ith region of the file when it is divided into 16 regions. Without collective I/O, each process should access its own tile data in the file. As shown in Figure 10, however, rows in each tile are distributed across the whole file, and thus accessing a tile will be noncontigu-

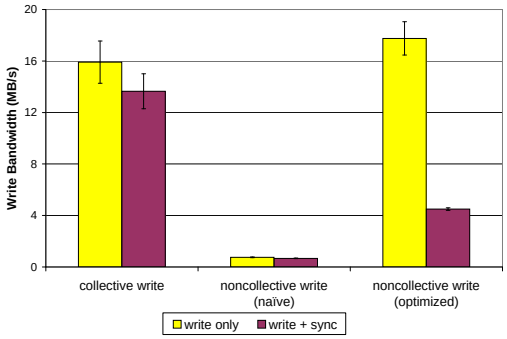

Figure 11. Tile write performance.

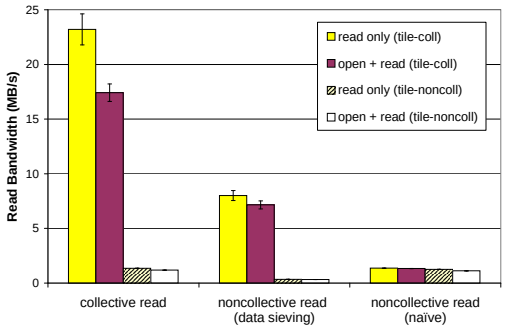

Figure 12. Tile read performance.

ous. But, unlike collective I/O, communication for data reorganization among processes will not be required.

Figure 11 presents the tile write performance between 16 Jazz nodes and the same IBP depot used for previous experiments, with a single thread and 4 MB block size. Since many simulations store different data sets in different files, and sync cost is visible with each file close, we show both the bandwidth with and without sync. With collective writes, the data is first shuffled among 16 processes, and then each process concurrently issues a 4 MB contiguous write request, resulting in 16 mappings and allocations. Even though the local communication cost is included, the collective write performs better than the single process write with 16 threads shown earlier, because the data transfer is performed via multiple network interface cards and also it does not incur the context switching costs. With the sync cost included, the collective write bandwidth drops by about 14%. Since the sync cost depends on the number of mappings in the exNode, if we collectively write larger data with the same number of mappings by increasing the block size, the performance drop caused by the sync is expected to decrease more.

With the naïve noncollective writes, performance drops significantly, because each process issues 1024 remote write requests (1024 rows in the subarray), and thus the overhead for multiple write requests adds up and hurts the performance. This will create 16384 mappings in the exNode, and the resulting XML file will be over 16 MB, 1024 times larger than the one generated for collective write. This large exNode significantly increases the sync cost. Indeed, in our experiments, the sync for noncollective writes took more than twice as long as the collective write and sync cost for the same data set.

With the optimized approach, the write-only bandwidth becomes about 12% higher than that of the collective write because each process performs only one write like the collective write, but local communication cost is not involved. Also, the cost to create 1024 mappings on each process is negligible. With the sync, however, the bandwidth becomes almost one-fourth of the write-only bandwidth, because the same number of mappings are created as with the naïve approach; thus, sync becomes expensive for the exNode with 16384 mappings.

Therefore, even with slightly worse write-only performance than optimized noncontiguous writes, collective write seems to be a clear winner among the three approaches, because it reduces the per-operation overhead and the sync cost by issuing fewer writes and thus creating fewer mappings. More benefits of collective write are described below.

The tile read performance was evaluated from two remote files, one written collectively (`tile-coll`) and the other written noncollectively with optimized noncontiguous writes (`tile-noncoll`). Both consist of 16 allocations. The former contains one mapping to each allocation, while the latter contains 1024 mappings to each.

Figure 12 shows the tile read performance with and without file open cost, to illustrate the effect of opening a file with a large number of mappings. The sync cost is negligible for read-only files and thus not included. The graph shows that the collective read performance with `tile-coll` yielded the best performance among the configurations used, because collective read issues fewer, larger remote read requests and thus reduces the per-operation overhead. With the open cost, the bandwidth drops about 25%. As mentioned, since the open cost depends on the number of mappings in the exNode, the gap between the two bars will decrease if we read larger data with the same number of mappings.

Compared to the collective read with `tile-coll`, however, the collective read with `tile-noncoll` decreased the performance by a factor of 17. The reason is that even though collective read issues larger reads, each read request is translated into many more small reads because of the way `tile-noncoll` was written. Also, the open costs more than twice the collective read cost of `tile-coll`. This again confirms why collective writes can improve I/O performance in MPI-IO/L, in this case by reducing the number of mappings in the exNode. The way a file was written affects not only the sync performance but also future open and read performance. When `tile-coll` was read noncollectively with data sieving, only about one-third of the collective read bandwidth was achieved, because of the excessive reads. In our configuration, the size of each tile is 4 MB, but the data

sieving reads almost 16 MB of data on each process. Moreover, when the whole array is read with the data sieving, each tile is read almost four times, while the collective read avoids this redundancy. Thus, although data sieving issues larger read requests, it is not always a desirable option for distributed global array reads, especially with slow remote I/O, and hence its use should be restricted for true noncontiguous I/O patterns. Reading `tile-coll` noncollectively with the naïve approach performs even worse than the data sieving. The naïve approach issues 1024 remote read requests on each process, and the per-operation overhead for small reads hurts the performance too much. With the open cost, the performance drops about 10%.

When `tile-noncoll` is read noncollectively, however, the data sieving performs much worse than the naïve approach. The naïve approach issues the same number of remote read requests as for reading `tile-coll`, resulting in similar performance. However, since the data sieving reads extra data and these extra read requests are translated into many small reads because of the way the file was written, more overhead is incurred. In our configuration, almost four times more read requests are issued for the data sieving than for the naïve approach, which issues 1024 read requests on each process. We note that even though larger read requests are made, they could issue many small read requests according to how the file was originally written.

For higher-dimensional data, this performance discrepancy between reading collectively and noncollectively written files is likely to increase because, with higher dimensionality, more complex noncontiguous access patterns that contain many more contiguous regions are likely. Thus, the benefit of reducing the number of mappings becomes even more important.

4 Discussion

Logistical Networking was not originally designed for partial file access or parallel I/O. Typical users run command line tools to access remote files, and the original design works well for such uses. For parallel I/O, however, its current design imposes limitations that might seriously affect the performance. This section discusses such issues and suggests how we can improve Logistical Networking to make it work better with parallel I/O systems.

Downsizing exNodes. LoRS currently creates a new allocation and a mapping to it for each write. Thus, fine-grained writes or frequent updates to the file could result in very large exNodes. With larger exNodes, open, sync, and close, which access the XML representation of exNodes, become more time-consuming. Large exNodes also take more memory on each process when stored in in-memory structures and thus reduce the amount of memory that can be otherwise used for storage and buffering of data. Moreover, it is not convenient to share such bulky exNodes with other people.

Multiple options are available for downsizing the exNodes. First, applications can generate fewer mappings by observing access patterns and eagerly using the available optimizations. For example, buffered I/O and collective I/O coalesce small I/O requests and thus reduce the number of actual writes. These approaches will also reduce the overhead associated with each I/O call.

One can also change the in-file exNode representation by rearranging the order of fields appearing in the file. Currently, each mapping is listed separately, regardless of the allocation it points to. With our optimized noncontiguous writes, however, multiple mappings can point to the same allocation. Mappings to the same allocation have identical values for certain fields, such as the three long capability keys, and with the current representation, these values are repeated. A better way to avoid such redundancy is to introduce a hierarchy between allocations and mappings so that for each allocation, allocation-related information such as capabilities appears first followed by the list of mappings associated with the allocation. In addition to reducing the exNode size, this approach provides an opportunity for noncontiguous read optimization, as described below.

Other possibilities include the implementation of the write-at-offset IBP that will simplify overwrites and an off-line data rearrangement tool that contiguously rewrites noncontiguously stored data to new allocations.

Efficient Noncontiguous I/O Support. The optimized noncontiguous writes pack and write contiguous regions in a noncontiguous extent into a single allocation. But, because of the limitation of the current exNode representation, the future reads whose extents overlap with such noncontiguously written extent cannot detect this fact and are not optimized properly. They issue separate small read requests to the same allocation, rather than reading the whole allocation at once.

If the exNodes are organized as proposed above where mappings that point to the same allocation are grouped under the allocation information, read could be further optimized as follows. First, we identify the allocations that overlap with the requested read extent. Next, for each such allocation, we decide to perform either contiguous or noncontiguous read by examining the mappings that point to the allocation. If noncontiguous read is detected, data sieving could be performed within that allocation for better performance. Since this approach performs data sieving on each allocation, instead of for the whole noncontiguous extent, it could reduce the amount of extra data read significantly.

If noncontiguous I/O is supported at the IBP level, further optimization is possible. For example, list I/O [15] describes a noncontiguous I/O access using a single

I/O interface, reducing the per I/O-call overhead. Similarly, datatype I/O [7, 10] provides an interface where a noncontiguous I/O pattern is described using an MPI-derived datatype. Once the underlying file system understands such advanced interfaces, it can better optimize the noncontiguous I/O performance.

5 Related Work

A few efforts have provided remote I/O through MPI-IO, all using ROMIO as the testbed. RIO [9] provided a preliminary design and proof-of-concept implementation of remote I/O in ROMIO. RIO used the ADIO layer for portability and later work, including MPI-IO/L, followed the same approach. However, RIO required a certain processor configuration that could cause inefficiency and relied on a legacy communication protocol. RFS [10] is a recent work that removed RIO's shortcomings. RFS seeks to reduce the apparent write cost by overlapping writes with subsequent computation phases through aggressive memory buffering. Both RIO and RFS adopt a client-server architecture, where a remote I/O request from the client is shipped to the server and executed there. On the other hand, MPI-IO/L translates each I/O request into LoRS calls and relies on Logistical Networking to take care of data transfer and storage. Remote I/O using GridFTP for ROMIO [3] uses a similar approach.

Like Logistical Networking, specialized data transfer and storage services such as GridFTP [2], Kangaroo [14], and GASS [6] can be used as a means of wide-area data transfer for I/O libraries. These mechanisms provide useful remote I/O features such as secure communication, a chainable server architecture, and workload-specific I/O optimizations. A metadata catalog system such as the MCAT in the Storage Resource Broker (SRB) [4], which is used to identify and discover resources and data sets of interest using their attributes instead of physical file names, is another useful feature to have in an I/O system. For Logistical Networking, a simple Web-based catalog system called Logistical Distribution Network (LoDN) was built recently. The use of LoDN together with MPI-IO/L will further improve the usability of the system.

6 Conclusions

We have presented the design and implementation of MPI-IO/L, an efficient remote I/O system for MPI-IO using Logistical Networking. Leveraging Logistical Networks, MPI-IO/L enables high-performance remote I/O and provides a flexible way to describe and share remote files. Our implementation with ROMIO provides various options for basic I/O operations so that the user can choose the I/O methods that work the best for her own application's I/O needs. We have also identified a number of areas in which Logistical Networking could be improved to better suit the needs of parallel I/O.

References

[1] NCSA HDF home page. http://hdf.ncsa.uiuc.edu.

[2] B. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke. Data management and transfer in high performance computational grid environments. *Parallel Computing Journal*, 28(5):749–771, 2002.

[3] T. Baer and P. Wyckoff. A parallel I/O mechanism for distributed systems. In *Proceedings of the International Conference on Cluster Computing*, 2004.

[4] C. Baru, R. Moore, A. Rajasekar, and M. Wan. The SDSC storage resource broker. In *Proceedings of the IBM Centers for Advanced Studies Conference*, 1998.

[5] M. Beck, T. Moore, and J. Plank. An end-to-end approach to globally scalable network storage. In *Proceedings of SIGCOMM*, 2002.

[6] J. Bester, I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke. GASS: A data movement and access service for wide area computing systems. In *Proceedings of the Workshop on Input/Output in Parallel and Distributed Systems*, 1999.

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

[8] I. Foster and C. Kesselman, editors. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2003.

[9] I. Foster, D. Kohr, Jr., R. Krishnaiyer, and J. Mogill. Remote I/O: Fast access to distant storage. In *Proceedings of the Workshop on Input/Output in Parallel and Distributed Systems*, 1997.

[10] J. Lee, X. Ma, R. Ross, R. Thakur, and M. Winslett. RFS: Efficient and flexible remote file access for MPI-IO. In *Proceedings of the International Conference on Cluster Computing*, 2004.

[11] J. Li, W.-K. Liao, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netCDF: A high-performance scientific I/O interface. In *Proceedings of SC03*, 2003.

[12] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Standard*. 1997.

[13] J. Plank, S. Atchley, Y. Ding, and M. Beck. Algorithms for high performance, wide-area distributed file downloads. *Parallel Processing Letters*, 13(2):207–224, 2003.

[14] D. Thain, J. Basney, S.-C. Son, and M. Livny. The Kangaroo approach to data movement on the Grid. In *Proceedings of the Symposium on High Performance Distributed Computing*, 2001.

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

[16] R. Thakur, R. Ross, and R. Latham. Implementing byte-range locks using MPI one-sided communication. In *Proceedings of the European PVM/MPI Users' Group Meeting*, 2005.