Transparent Log-Based Data Storage in MPI-IO Applications

Dries Kimpe[1,2], Rob Ross[3], Stefan Vandewalle[1], and Stefaan Poedts[2]

[1] Technisch-Wetenschappelijk Rekenen, K.U.Leuven, Celestijnenlaan 200A, 3001 Leuven, België
Dries.Kimpe@cs.kuleuven.be
[2] Centrum voor Plasma-Astrofysica, K.U.Leuven, Celestijnenlaan 200B, 3001 Leuven, België
[3] Argonne National Laboratory, 9700 S Cass Ave, 60439 Argonne, IL

Abstract. The MPI-IO interface is a critical component in I/O software stacks for high-performance computing, and many successful optimizations have been incorporated into implementations to help provide high performance I/O for a variety of access patterns. However, in spite of these optimizations, there is still a large performance gap between "easy" access patterns and more difficult ones, particularly when applications are unable to describe I/O using collective calls.
In this paper we present LogFS, a component that implements log-based storage for applications using the MPI-IO interface. We first discuss how this approach allows us to exploit the temporal freedom present in the MPI-IO consistency semantics, allowing optimization of a variety of access patterns that are not well-served by existing approaches. We then describe how this component is integrated into the ROMIO MPI-IO implementation as a stackable layer, allowing LogFS to be used on any file system supported by ROMIO. Finally we show performance results comparing the LogFS approach to current practice using a variety of benchmarks.

1 Introduction

Dealing with complex I/O patterns remains a challenging task. Despite all optimizations, there is still a huge difference in I/O performance between simple (contiguous) and complex (non-contiguous) access patterns [2] This difference can be attributed to physical factors (non-contiguous patterns typically cause read-modify-write sequences and require time-consuming seek operations) and to software issues (complex patterns require more memory and processing). Recent papers concentrated on reducing the software overhead associated with processing complex I/O patterns [4–6].

In computational science *defensive I/O* is common: applications write checkpoints in order to provide a way to rollback in the event that a system failure terminates the application prematurely. In the event that an application successfully executes for an extended period, these checkpoints may never actually be read. So while the application might exhibit complex access patterns during checkpoint write, if we can defer the processing of the complex pattern, we might attain much higher throughputs than otherwise possible. Of course, some mechanism must be available for post-processing the complex pattern in the event that the data is read, ideally with little or no additional overhead. If this post-processing mechanism is fast enough, this approach could be used as a general-purpose solution as well.

The MPI-IO interface [9] is becoming the standard mechanism for computational science applications to interact with storage, either directly or indirectly via high-level libraries such as HDF5 [10] and Parallel netCDF [11]. This makes the MPI-IO implementation an ideal place to put I/O optimizations. The MPI-IO default consistency semantics are more relaxed than the traditional POSIX semantics [12] that most file systems strive to implement. In particular, the MPI-IO default semantics specify that only local changes are visible to an application process until an explicit file synchronization call is made. This aspect of the standard will enable us to further optimize our implementation.

In this paper we present LogFS, an extension to the ROMIO MPI-IO implementation [1] designed to provide log-based storage of file data in parallel applications. The functionality is provided transparently to the user, it follows the MPI-IO consistency semantics, and a mechanism is provided for reconstituting the canonical file so that UNIX application can subsequently access the file. In Section 2 we describe the LogFS approach. In Section 3 we show how this approach has significant performance benefits in write-heavy workloads. In Section 4 we conclude and discuss future directions for this work.

1.1 Related work

Two groups are actively pursuing research in MPI-IO optimizations that are relevant to this work. The group at Northwestern University has been investigating mechanisms for incorporating cooperative caching into MPI-IO implementations to provide what is effectively a large, shared cache. They explore using this cache for both write-behind, to help aggregate operations, and to increase hits in read-heavy workloads [15, 16]. Our work is distinct in that we allow our "cache" to spill into explicit log files, and we perform this caching in terms of whole accesses rather than individual pages or blocks. Their implementations to date have also relied on the availability of threads, MPI-2 active-target one-sided communication, or the Portals interface, limiting applicability until active-target operations become more prevalent on large systems. Our work does not have these requirements.

Yu et. al. have been investigating mechanisms for improving parallel I/O performance with the Lustre file system. Lustre includes a feature for joining previously created files together into a new file. They leverage these features to create files with more efficient striping patterns, leading to improved performance [14]. Their observations on ideal stripe widths could be used to tune stripe widths for our log files or to improve the ration of creation of the final canonical file on systems using Lustre.

2 LogFS

The LogFS extension to ROMIO provides independent, per-process write logging for applications accessing files via MPI-IO. On each process, logging is separated into a log file containing the data to be written, called the *datalog* and a log file holding metadata about these writes such as location and epoch, called the *metalog*. A global *logfs file* is stored while the file is in log format and maintains a list of the datalogs and metalogs. Together these files maintain sufficient data that the correct contents of the file can be generated at any synchronization point.

The logfs file is initially created using `MPI_MODE_EXCL` and `MPI_MODE_CREATE`. Note that in a production implementation we would need to manage access to this file so that the file was open by only one MPI application if logging was in progress. This could be managed using an additional global lockfile.

The datalog and metalog files are opened independently with `MPI_COMM_SELF` and are written sequentially in contiguous blocks, regardless of the application's write pattern, hiding any complexity in the write pattern and deferring the transformation of the data into the canonical file organization (i.e. the traditional POSIX file organization) until a later time.

The data logfile contains everything written to the file by the corresponding process. Data is written in large contiguous blocks corresponding to one or more MPI-IO write operations. This lends itself to high performance on most parallel file systems, because there is no potential for write lock contention.

The metadata logfile records, for every write operation by the process, the file offset (both in the real file and in the datalog) and the transfer size. In addition to write operations, the metadata log also tracks `MPI_File_sync`, `MPI_File_set_size` and `MPI_File_set_view`. However, this is done in a lazy fashion: only changes actually needed to accurately replay the changes are stored. Calling sequences without effect to the final file, such as repeatedly changing the file view without actually writing to the file, are not recorded in the metalog. Of these operations, all have a fixed overhead, except for `MPI_File_set_view`. Currently, datatypes are stored as lists of $\langle$*offset, size*$\rangle$ pairs.

It was observed in early parallel I/O studies that parallel applications often perform many, small, independent I/O operations [13]. This type of behavior continues today, and in some cases high-level I/O libraries can contribute through metadata updates performed during I/O. Keeping this in mind, LogFS can additionally use a portion of local memory for aggregation of log entries. This aggregation allows LogFS to more efficiently manage logging of I/O operations and convert many small I/O operations into a fewer number of larger contiguous ones, again sequential in file.

2.1 Creating the Canonical File for Reading

By default, LogFS tries to postpone updating the canonical file for as long as possible. In some situations, such as files opened in write-only mode, even closing the file will not necessarily force a replay. This allows for extremely efficient checkpointing.

When a LogFS file is opened in `MPI_MODE_RDONLY`, the canonical file is automatically generated if logs are still present. The canonical file is generated through a collective "replay" of the logs. In our implementation we assume that the number of replay processes is the same as the original number of writer processes, but replay processes could manage more than one log at once.

Replay occurs in *epochs* corresponding to writes that occur between synchronization points. By committing all writes from one epoch before beginning the next, we are able to correctly maintain MPI-IO consistency semantics without tracking the timing of individual writes. Each process creates an in memory rtree[8], a spatial data structure allowing efficient range queries. Replayers move through the metalog, updating their rtree with the location of the written data in the datalog. When the replayer hits the end of an epoch or the processed data reaches a certain configurable size, the replayer commits these changes to the canonical file. To accomplish a commit, the replayer process reads the data from the datalog into a local buffer, calls `MPI_File_set_view` to define the region(s) in the canonical file to modify, then calls `MPI_File_write_all` to modify the region(s). Processes use `MPI_Allreduce` between commits to allow all processes to complete one epoch before beginning the next. This approach to replay always results in large, collective I/O operations, allowing any underlying MPI-IO optimizations to be used to best effect [7].

This also enables the user to only replay those files that are actually needed. With this system, LogFS is transparent to all applications using MPI-IO; replay will happen automatically when needed. However, it is often the case that post-processing tools are not written using MPI-IO. In the climate community, for example, Parallel netCDF is often used to write datasets in parallel using MPI-IO, but many post-processing tools use the serial netCDF library, which is written to use POSIX I/O calls. For those situations, a small stand-alone utility is provided that can force the replay of a LogFS file.

An additional MPI-IO hint, `replay-mode` is understood by the LogFS-enabled ROMIO. When this is set to "replay-on-close", replay is automatically performed when a LogFS file is closed after writing. The stand-alone tool simply opens the LogFS file for writing with this hint set, then closes the file. Note that with this approach as many processes as originally wrote the LogFS file may be used to replay in parallel.

2.2 Mixed Read and Write Access

The LogFS system is obviously designed for situations where writes and reads are not mixed. However, for generality we have implemented two mechanisms for supporting mixed read and write workloads under LogFS.

When `MPI_MODE_RDRW` is selected for a file, MPI consistency semantics require that a process is always able to read back the data it wrote; Unless atomic mode is also enabled, data written by other processes only has to become visible after `MPI_File_sync` is called (or after closing and re-opening the file, which performs an implicit sync). If a user chooses both atomic mode and `MPI_MODE_RDRW`, LogFS optimizations are not appropriate, and we will ignore that case in the remainder of this work.

In order to guarantee that data from other processes is visible at read time, we replay local logs on each process at synchronization points. Replay consists of local independent reads of logs followed by collective writes of this data in large blocks. This has the side-effect of converting all types of application access (independent or collective, contiguous or noncontiguous) into collective accesses, increasing performance accordingly [7].

We have two options for guaranteeing that data written locally is returned on read operations prior to synchronization. A simple option is for processes to ensure that the canonical file is up-to-date on read; this may be accomplished by performing a local replay in the event of a read operation. In this case, only the first read operation will be slow, all subsequent reads will continue at native speeds. However, this method performs badly with strongly mixed I/O sequences; Frequent reads force frequent log replays, and the efficiency of write aggregation diminishes with increased replay frequencies.

Another option is for each process to track regions written locally since the last sync operation. If those regions overlap with parts of a read request, data needs to be read from the datalog. Accesses to unchanged regions may be serviced using data from canonical file. If a very large number of writes occur, and the memory cost of tracking each individual region becomes too high, we have the option of falling back to our first option and completely replaying the local log, removing the need to track past regions.

To efficiently track write regions during `MPI_MODE_RDRW` mode, every process maintains an in-memory rtree at run-time. For every written region, the rtree records the location of that data in the datalog of the process, and on every write operation this rtree is updated, in a manner similar to the approach used in replay.

The rtree then provides us with an efficient mechanism for determining if local changes have been made since the last synchronization, and if so, where that data is located in the datalog. With this scheme a read request gets transformed by LogFS into at most two read operations; one to read data from the datalog, and one to read any remaining data from the canonical file.

Unfortunately, tracking all affected regions in long-living files with lots of fragmented write accesses can lead to large rtree descriptions. In these cases we are forced to either update the canonical file with local changes, to shrink the rtree, or stop tracking writes all together and fall back to our original option.

2.3 Implementing in ROMIO

LogFS is implemented as a component integrated into ROMIO[1]. ROMIO incorporates an interface for supporting multiple underlying file systems called ADIO. We prototyped two approaches for implementing LogFS.

LogFS ADIO Implementation The first approach was to implement LogFS as a new ADIO component. In this approach LogFS appears as a new file system type, but internally it makes use of some other ADIO implementation for performing file I/O. For example, a user opening a new file on a PVFS file system using the "logfs:" prefix on their file name would create a new LogFS-style file with logs and canonical file stored on the PVFS file system. A consequence of this approach is that LogFS can be enabled on any filesystem supported by ROMIO, and the file may be written in the usual "normal" data representation.

Under a strict interpretation of the MPI-IO standard, changes to a file in the "normal" (or "external32") data representation must be made visible to other applications at synchronization points, unless the file is opened with `MPI_MODE_UNIQUE_OPEN`. To meet this strict interpretation of the standard, LogFS must perform a full replay at synchronization points if unique open is not specified, even when in write-only mode. This approach is only most effective when applications use the unique open mode.

LogFS as a Data Representation Our second approach was to implement LogFS as ROMIO's "internal" data representation, a somewhat creative interpretation of the internal data representation specification. To function as a data representation, LogFS must intercept all file access operations. For this purpose, a layering technique for ADIO components was developed which allowed transparent interception of all ADIO methods.

When the user changes to our internal data representation (using `MPI_File_set_view`, the LogFS ADIO is layered on top of the active ADIO driver for the file. One difference between this approach and the ADIO approach is that the data representation may be changed through `MPI_File_set_view` at any time, so if the view is later restored to its original setting, the logfiles are immediately replayed and the canonical file created.

According to the standard, the format of a file stored in the internal data representation is not known. This means that we can force applications to open the file and change the data representation back to "native" prior to access by application not using MPI-IO. This hook allows us to avoid the need to replay logs at synchronization points in the general case.

3 Performance Results

In this section we show results of experiments comparing the LogFS approach to a stock ROMIO implementation (included in mpich 1.0.5p4). As the base filesystem, PVFS[17] version 2.6.3 was used. The filesystem was configured to use TCP (over gigabit ethernet) as network protocol, with 4 I/O servers and 1 metadataserver. For testing the filesystem, 16 additional nodes were used.

Before we present our results we will first quantitatively describe the overhead incurred by our logging process.

3.1 Overhead

There is a certain amount of overhead introduced by first recording I/O operations in the logfiles. During the write-phase, overhead consists of meta data (describing the I/O operation) and actual data (the data to be written), both stored in the logfiles. Likewise, during replay, everything read from the logfiles can be considered overhead. Table 1 indicates per-operation overhead based on the storage format described in 2.

Operation	Log File Overhead	
	Metalog	Datalog
`MPI_File_write`	datalog offset, write offset (2x `MPI_OFFSET`)	*datatype size* × *count*
`MPI_File_set_view`	displacement(`MPI_OFFSET`) + etype(*flatlist*) + filetype(*flatlist*)	0
`MPI_File_sync`	epoch number(`MPI_INT`)	0
`MPI_File_set_size`	filesize(`MPI_OFFSET`)	0

Table 1. LogFS logfile overhead.

Many scientific applications perform regular checkpointing. Typically, only the latest or a small number of checkpoints are kept; In this case, most of the data written to the checkpoint file will never survive; it will be over-written during run-time or deleted shortly after the the application terminates. Although all data will be forced out to the datalog, in the event that data is over-written it will never be read again because a replay of the metalog will only keep track of the most recent data.

In the worst case when data is not overwritten, all data will be read again during replay. However, since data in the metalog is accessed sequentially and in large blocks, these transfers typically reach almost full filesystem bandwidth.

3.2 Results

For testing, we choose the well known "noncontig" benchmark. Noncontig partitions the test file in vectors of a fixed size, allocating them in a round-robin fashion to every CPU. This generates a non-contiguous regular strided access pattern. Figure 1 shows how the LogFS write bandwidth compares to that of a stock ROMIO implementation.

The results clearly show how LogFS is capable of transforming extremely inefficient access patterns (such as the independent non-contiguous pattern in

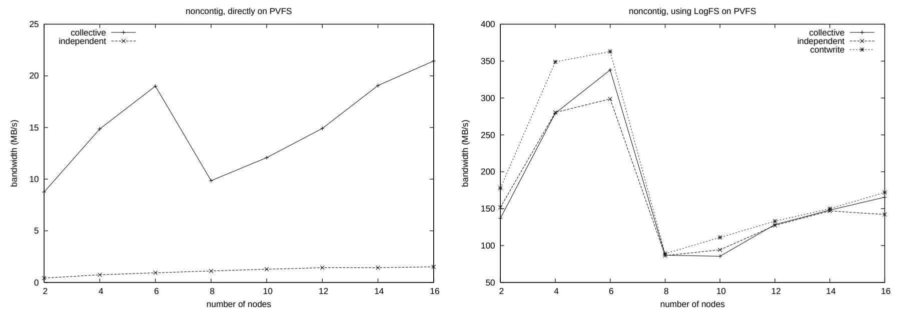

Fig. 1. noncontig benchmark (vector count = 8388608)

the left figure) into faster ones. Using LogFS, a peak write bandwidth of approx. 300 MB/s is reached. Without LogFS, peak bandwidth is only approx. 20 MB/s for collective access, and only around 2 MB/s for independent access.

To see how closely LogFS gets to the maximum write bandwidth possible, a small test program ("contwrite") was created that directly writes large blocks – the same size of the write-combining buffers used in two-phase and LogFS – to the filesystem. The results can be seen in Figure 1.

(Additional results detailing replay performance will be added in a later version of this paper)

4 Conclusions and Future Work

LogFS is capable of enhancing write performance of programs using complex file access patterns. Its layered design and modest file system requirements allow the approach to be employed on a wide variety of underlying file systems. For application checkpoints that might never be read again, performance can be improved by at least a factor of 10.

LogFS is primarily targeted at write-only (or write-heavy) I/O workloads. However, much of the infrastructure implemented for LogFS may be reused to implement independent per-process caching in MPI-IO. We are actively pursuing this development. This approach provides write aggregation benefits, can transform a larger fraction of I/O operations into collective ones, and has benefits for read-only and read-write workloads. Similar to LogFS, per-process caching doesn't require MPI threads or active-target one-sided MPI operations, meaning that it can be implemented on systems such as the IBM Blue Gene/L [3] and Cray XT3 that lack these features.

Currently, LogFS creates one logfile for every process opening the file. When running on large numbers of processes, this leads to a huge amount of logfiles. To avoid this, we are considering sharing logfiles between multiple processes. This approach is complicated by the need for processes opening files in read/write mode to "see" local changes between synchronization points, because this means that multiple processes might need to read the same log files at runtime.

Currently, datatypes are stored as lists of $\langle$*offset, size*$\rangle$. By using the `MPI_Type_get_envelope` function, a more compact description using type constructors could be used instead. This would further reduce the amount of meta-data that needs to be logged.

References

1. Thakur, R., Gropp, W., and Lusk, E., "An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces," in Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation, October 1996, pp. 180-187.
2. Kimpe, D., Vandewalle, S., Poedts, S. "On the Usability of High-Level Parallel IO in Unstructured Grid Simulations", in Proceedings of the 13th EuroPVM/MPI Conference, September 2007, pp. 400-401.
3. Allsopp, N., Follows, J., Hennecke, M., Ishibashi, F., Paolini, M., Quintero, D., Tabary, A., Reddy, H., Sosa, C., Prakash, S. and Lascu, O., Unfolding the IBM Eserver Blue Gene Solution, International Business Machines Corporation, September, 2005.
4. Worringen, J., Traff, J., and Ritzdorf, H., "Improving Generic Non-Contiguous File Access for MPI-IO," in Proceedings of the 10th EuroPVM/MPI Conference, September, 2003.
5. Ross, R., Miller, N., and Gropp, W., "Implementing Fast and Reusable Datatype Processing," in Proceedings of the 10th EuroPVM/MPI Conference, September, 2003.
6. Hastings, A., and Choudhary, A., "Exploiting Shared Memory to Improve Parallel I/O Performance", in Proceedings of the 13th European PVM/MPI Users' Group Meeting (Euro PVM/MPI 2006), September, 2006.
7. Thakur, R., Gropp, W., and Lusk, E. 1998, "A case for using MPI's derived datatypes to improve I/O performance," in Proceedings of the 1998 ACM/IEEE Conference on Supercomputing, San Jose, CA, November, 1998.
8. Guttman, A., "R-Trees: A Dynamic Index Structure for Spatial Searching," in Proceedings of the ACM International Conference on Management of Data (SIGMOD), 1984.
9. The Message Passing Interface Forum, "MPI-2: Extensions to the Message-Passing Interface," July, 1997.
10. HDF5,`http://hdf.ncsa.uiuc.edu/HDF5/`.
11. Li, J., Liao, W., Choudhary, A., Ross, R., Thakur, R., Gropp, W., Latham, R., Siegel, A., Gallagher, B., and Zingale, M., "Parallel netCDF: A High-Performance Scientific I/O Interface", in Proceedings of SC2003, November 2003.
12. IEEE/ANSI Std. 1003.1, "Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API) C Language," 1996.
13. Purakayastha, A., Ellis, C., Kotz, D., Nieuwejaar, N., and Best M., "Characterizing Parallel File-Access Patterns on a Large-Scale Multiprocessor," in Proceedings of the Ninth International Parallel Processing Symposium, April, 1995.
14. Yu, W., Vetter, J., Canon, R., and Jiang, S., "Exploiting Lustre File Joining for Effective Collective IO," Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2007), May 2007.
15. Coloma, K., Choudhary, A., Liao, W., Ward, L., and Tideman, S., "DAChe: Direct Access Cache System for Parallel I/O," In the 2005 Proceedings of the International Supercomputer Conference, June, 2005.

16. Liao, W., Ching, A., Coloma, K., Choudhary, A., and Kandemir, M., “Improving MPI Independent Write Performance Using A Two-Stage Write-Behind Buffering Method,” In the Proceedings of the Next Generation Software (NGS) Workshop, held in conjunction with the 21th International Parallel and Distributed Processing Symposium (IPDPS), Long Beach, California, March 2007.
17. P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur, “PVFS: A Parallel File System For Linux Clusters” In the Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta, GA, October 2000, pp. 317-327.