Improving Collective I/O Performance Using Non-Volatile Memory Devices

Giuseppe Congiu*, Sai Narasimhamurthy*, Tim Süß†, André Brinkmann†
*Emerging Technology Group Seagate Systems Ltd Havant, United Kingdom
Email: {giuseppe.congiu, sai.narasimhamurthy}@seagate.com
†Zentrum für Datenverarbeitung Johannes Gutenberg-Universität Mainz, Germany
Email: {t.suess, brinkman}@uni-mainz.de

***Abstract*—Collective I/O is a parallel I/O technique designed to deliver high performance data access to scientific applications running on high-end computing clusters. In collective I/O, write performance is highly dependent upon the storage system response time and limited by the slowest writer. The storage system response time in conjunction with the need for global synchronisation, required during every round of data exchange and write, severely impacts collective I/O performance. Future Exascale systems will have an increasing number of processor cores, while the number of storage servers will remain relatively small. Therefore, the storage system concurrency level will further increase, worsening the global synchronisation problem. Nowadays high performance computing nodes also have access to locally attached solid state drives, effectively providing an additional tier in the storage hierarchy. Unfortunately, this tier is not always fully integrated. In this paper we propose a set of MPI-IO hints extensions that enable users to take advantage of fast, locally attached storage devices to boost collective I/O performance by increasing parallelism and reducing global synchronisation impact in the ROMIO implementation. We demonstrate that by using local storage resources, collective write performance can be greatly improved compared to the case in which only the global parallel file system is used, but can also decrease if the ratio between aggregators and compute nodes is too small.**

keywords*—*MPI-IO; Collective I/O; Non-Volatile Memory; HPC;

I. Introduction

High Performance Computing (HPC) applications process large amounts of data that have to be written (read) to (from) large shared files residing in the global parallel file system. In order to make the data set manageable, this is usually partitioned into smaller sub-sets and assigned to available cores for parallel processing. Complex data sets such as N-dimensional arrays are logically flattened into a linear sequence of bytes and striped across several I/O targets for best performance. This results in the loss of the original spatial locality. Due to this characteristic, accesses to spatially contiguous regions translate into non-contiguous accesses to the file. Therefore, applications generating a large number of small, non-contiguous I/O requests to the parallel file system usually experience degradation of I/O performance. Such performance degradation is known as the 'small I/O problem' and is related to the fact that parallel file systems provide best I/O bandwidth performance for large contiguous requests, while they typically provide only a fraction of the maximum available bandwidth in the opposite case [14] [17]. This is due to the large number of Remote Procedure Calls (RPCs) generated by the file system clients overwhelming I/O servers, the resulting high number of hard disk head movements in every I/O target (seek overhead) and, ultimately, to the restrictions imposed by the POSIX I/O write semantic that generates lock contention on file systems' blocks.

Having recognised the small I/O problem, collective I/O was proposed by the MPI-IO community [28]. Collective I/O exploits global I/O knowledge in parallel I/O to a shared file. This knowledge is used to build an aggregated view of the accessed region in the file and coalesce all the corresponding small non-contiguous requests into a smaller number of large contiguous accesses, later issued to the parallel file system. File system accesses are orchestrated in such a way that only a subset of the available processes actually performs I/O. These processes are called 'aggregators', because they gather and aggregate all the requests on behalf of the other processes, whose only role in this case is to send (receive) data to (from) aggregators.

In conclusion, collective I/O can convert many small random accesses into large sequential accesses to the I/O subsystem, reducing the actual number of I/O requests that need to be accounted for by the I/O stack. Even when processes generate large I/O requests, it might be still beneficial to coordinate them to reduce parallel file system block locking contention as well as concurrency level on data servers. This mechanism effectively adapts the I/O pattern to the characteristics of the file system, extracting maximum performance from it. Figure 1 exemplifies the basic collective I/O mechanism just described. In the figure there are four processes, two of which play the role of aggregators. Collective I/O proceeds in two phases: 'data shuffling' and 'data I/O'. Data shuffling takes place between all the processes and aggregators and is aimed to build the logically contiguous file regions (or file domains) that will be later accessed during the data I/O phase.

Two phase I/O has a number of problems that limit its scalability. The main issues are: (**a**) global synchronisation overhead between processes exchanging data, where I/O is bottlenecked by the slowest aggregator process [30], (**b**) aggregators' contention on file system stripes (primarily due to stripe boundary misalignment of the file domains and the underlying file system locking mechanism[1]) [20], (**c**) aggregators' contention on I/O servers (related to inefficient file domain partitioning strategy), (**d**) pressure that large collective buffers

[1]In current ROMIO versions the ADIO driver for Lustre can detect and align file domains to stripe boundaries thus avoiding stripe collisions. A new ADIO driver for BeeGFS supporting the same functionality has been developed in the course of this work.

can have on system memory (due to scarce amount of available memory per single core in current and future large scale HPC clusters), and (**e**) high memory bandwidth requirements due to the large number of data exchanges between processes and aggregators [23]. There is also a mismatch between logical file layouts and the collective I/O mechanism, which does not seem to take the physical data layouts into consideration [13] [31].

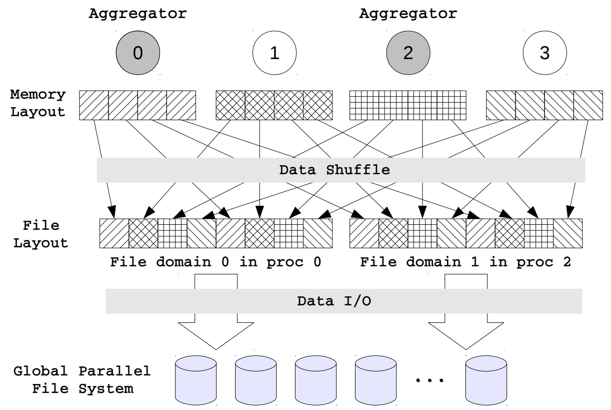

Fig. 1: Collective I/O write example. A data set is partitioned and assigned to four processes. Two of them work as aggregators writing data to the global file system.

Nowadays solid state drives (SSDs) are becoming more affordable and many HPC systems provide SSDs attached to compute nodes. Despite the fact that these devices formally provide an additional tier in the storage hierarchy, they are mostly only used to store local data and rarely exploited for parallel I/O. As part of the Exascale10 (E10) [4] initiative in the DEEP-ER [3] project, we explore the benefits that the usage of such fast devices can bring when writing collectively to a shared file.

The main contribution of this paper is the integration of local storage resources attached to compute nodes (SATA SSDs in our specific case, but this can be any non-volatile memory device) as persistent cache layer in ROMIO to buffer collective write data, providing all the additional infrastructure required to asynchronously flush the content of the cached data to the global file system, while keeping the MPI-IO consistency semantics. The new persistent cache layer is fully integrated inside ROMIO and can be easily accessed through a set of newly defined MPI-IO hints.

We show how when using local SSDs, if the number of aggregators is selected properly, significantly better I/O performance can be achieved compared to the case in which only the global file system is used. Contrarily, if the number of aggregators is not properly selected, I/O performance can even degrade. The additional cache layer also delivers more stable response times, reducing the impact of global synchronization on collective I/O, thus addressing point (**a**) previously discussed, and can reduce the memory pressure in systems with scarce amounts of main memory per core, thus addressing point (**d**).

The remainder of this paper is organised as follows. In Section II we describe in detail the collective I/O implementation in ROMIO highlighting all the possible bottlenecks, in Section III we present the integration of the caching layer and the new hints extensions supporting it, along with all the corresponding semantics implications, in Section IV we show the benefits of collective I/O using different benchmarks, in Section V we present collective I/O related works and finally in Section VI we draw conclusions and discuss future developments.

II. Collective I/O in ROMIO

ROMIO is a popular implementation of the MPI-IO specification developed at the Argonne National Laboratory and currently supported by MPICH as well as OpenMPI and other packages. ROMIO provides parallel I/O functionalities for different file systems through the Abstract Device I/O interface [27] (ADIO). Latest versions of ROMIO include support for Lustre [29], GPFS [25], PVFS [12] and others through a dedicated ADIO driver.

A. Two Phase I/O

In ROMIO, the core component of collective I/O is the 'two phase I/O', also known as 'extended two phase algorithm' (ext2ph) [26]. The ROMIO implementation for collective I/O consists of several steps as follows:

1) All processes taking part in the I/O operation exchange access pattern information with each other. The access pattern information is represented by start and end offsets for the accessed region (disregarding holes that may be present). Once file offsets are available, every process works out how big the global accessed region in the file is by taking maximum and minimum among all. The resulting byte range is divided by the number of available aggregators to build the so called 'file domains' (contiguous byte ranges accessed independently by every aggregator).
2) Every process works out which file domains (and thus aggregators) its local data belongs to. In doing so, every process knows which aggregators it has to send (receive) data to (from), if any.
3) Every aggregator works out which other processes' requests map to its file domain. Doing so every aggregator knows what processes need to receive (in case of reads) or send (in case of writes) data for that particular file domain.
4) Actual two phase I/O starts. In the case of writes, that we exclusively consider here (the read case is similar), every process sends its data to the right aggregators (data shuffle phase) while these write the data to the parallel file system (data I/O phase). Data is written in blocks of predefined size (collective buffer size). If the size of the collective buffer is smaller than the file domain, the file domain is broken down into multiple sub-domains which are written in different rounds of the ext2ph algorithm. In order to handle multiple rounds of data shuffle and I/O, additional access information is required. This is disseminated by every process (collectively) to aggregators at the beginning of the data shuffle phase.
5) Once all the data has been written, all the processes must synchronise and exchange error codes. This

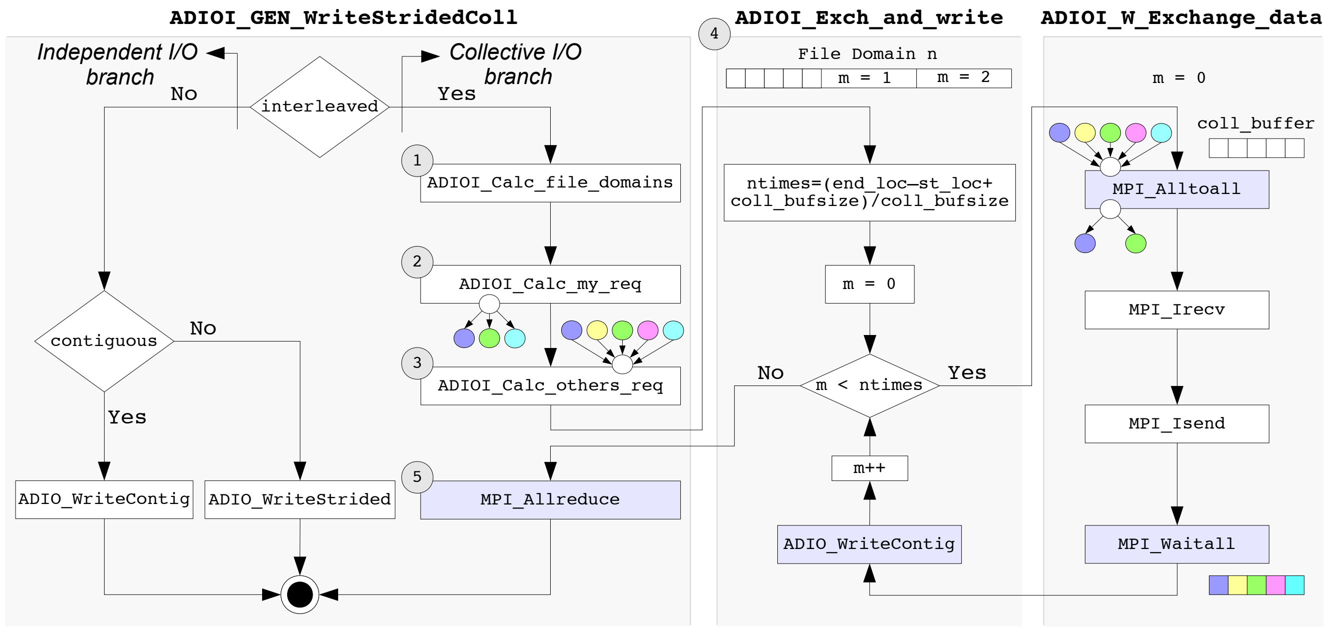

Fig. 2: Collective I/O flow diagram for the write path in aggregators (non-aggregators neither receive nor write any data, just send it to aggregators). `MPI_File_write_all()` invokes `ADIOI_GEN_WriteStridedColl()`. `ADIO_WriteContig` is a macro that is replaced by `ADIOI_GEN_WriteContig()`. Performance critical functions for the collective I/O branch are highlighted in grey.

is necessary to guarantee that it is safe to free the memory buffers containing the data.

Figure 2 shows how the previous steps map to the collective I/O implementation for the write operation. The collective write function (`MPI_File_write_all()`) in ADIO is implemented through `ADIOI_GEN_WriteStridedColl()`. This is responsible for selecting the most suitable I/O method between those available. For example, independent I/O is selected if the access requests are not interleaved. Nevertheless, users can always enforce collective I/O by setting the appropriate MPI-IO hint. The `ADIOI_Exch_and_write()` function contains the ext2ph algorithm implementation, including data shuffle and write methods. At the beginning of the data shuffle (`ADIOI_W_Exchange_data()`) we have the dissemination function (`MPI_Alltoall()`) used to exchange information concerning which part of the data has to be sent during a particular round of two phase I/O.

There are three main contributors to collective I/O performance: (**a**) global synchronisation cost; (**b**) communication cost; and (**c**) write cost. `MPI_Allreduce()` and `MPI_Alltoall()` account for the global synchronisation cost. When a process reaches them it has to wait for all the other processes to arrive before continuing. `MPI_Waitall()` accounts for communication cost since every process first issues all the non-blocking receives (if any) and sends, and afterwards waits for them to complete (refer to the right part of the diagram in Figure 2). Finally, `ADIO_WriteContig()` accounts for write cost.

B. *Collective I/O Hints*

Collective I/O behaviour can be controlled by users through a set of MPI-IO hints. Users can control whether collective I/O should be enabled or disabled with `romio_cb_write` and `romio_cb_read`, for write and read operations respectively, how many aggregators should be used during a collective I/O operation with `cb_nodes` and how big the collective buffer should be with `cb_buffer_size`. Table I summarises the hints just described.

TABLE I: Collective I/O hints in ROMIO.

Hint	Description
`romio_cb_write`	`enable` or `disable` collective writes
`romio_cb_read`	`enable` or `disable` collective reads
`cb_buffer_size`	set the collective buffer size [bytes]
`cb_nodes`	set the number of aggregator processes

Each of these hints has an effect on collective I/O performance. For example, by increasing the number of aggregators there will be a higher number of nodes writing to the parallel file system and thus a higher chance that one of these will experience variable performance due to load imbalance among available I/O servers, with increasing write time variation and associated global synchronisation cost. Furthermore, by increasing the collective buffer size users can reduce the number of two phase I/O rounds and, consequently, the number of global synchronisation events. Bigger collective buffers will also affect the write cost since more I/O servers will be accessed in parallel potentially increasing the aggregated I/O bandwidth.

Besides the hints described in Table I, there are other hints that do not directly concern collective I/O but affects its performance. The first is the `striping_factor` hint, which defines how many I/O targets will be used to store the file. The second is the `striping_unit` hint, which defines how big the data chunks written to each I/O target will be (in bytes). These two hints change the file characteristics in the parallel file system and typically the 'striping_unit' also defines the locking granularity for the file (e.g. Lustre).

III. MPI-IO HINTS EXTENSIONS

To the best of our knowledge, at the time of writing this paper, there is very little or no work on how to use non-volatile memory devices in computing nodes of an HPC cluster as persistent cache layer to boost collective I/O performance in ROMIO. The use of these devices can greatly increase parallelism, reduce write response time variations among processes and consequently global synchronisation cost. Data cached in locally attached SSDs can be synchronised independently by every aggregator in the background while the application can progress doing useful work, effectively converting collective I/O to independent I/O when writing to the parallel file system.

To take advantage of attached non-volatile memories in computing nodes we introduced a new set of MPI-IO hints, reported in Table II, and a corresponding set of modifications in the ROMIO implementation of the Universal File System (UFS) ADIO driver supporting them.

TABLE II: Proposed MPI-IO hints extensions.

Hint	**Value**
`e10_cache`	`enable`, `disable`, `coherent`
`e10_cache_path`	cache directory pathname
`e10_cache_flush_flag`	`flush_immediate`, `flush_onclose`
`e10_cache_discard_flag`	`enable`, `disable`
`ind_wr_buffer_size`	synchronisation buffer size [bytes]

The new hints are used to control the data path in the storage system as well as to define a basic set of cache policies for synchronisation and space management. In particular, the `e10_cache` hint is used to `enable` or `disable` the cache, directing applications' data to the local file system instead of the global file system. When the hint is set to `coherent` all the written data extents will be locked until cache synchronisation is completed. The `e10_cache_path` hint is used to control where in the local file system tree the cache file will reside. The `e10_cache_flush_flag` hint is used to control the synchronisation policy of cached data to the global file. If the hint is set to `flush_immediate` data will be immediately flushed to the global file. Alternatively, if the hint is set to `flush_onclose` data will be flushed to the global file when it is closed. The `e10_cache_discard_flag` hint is used to perform basic cache space management. In particular, if the hint is set to `enable` the cache file will be removed after the file is closed, otherwise (`disable`) it will be retained until the user manually removes it. Finally, the `ind_wr_buffer_size` hint controls the size of the buffer used to synchronise cached data to the global file. This hint already existed in ROMIO but was only used during independent I/O to determine the write granularity. The hints in Table II can be used in conjunction with the collective I/O hints described in Section II-B.

Besides the proposed cache policies, more complex ones are possible. For example, the cache synchronisation could take into account the level of congestion of the I/O servers. The cache replacement policy could also use a more complex strategy to evict cached files (or extents of data inside the file). Although these can be implemented in ROMIO, they introduce more sophisticated functionalities that go beyond the scope of this work.

A. *Cache Hints Integration in ROMIO*

As already mentioned, the introduced MPI-IO hints are supported by a corresponding set of modifications in the ROMIO implementation [5]. These modifications, following described, provide the functionalities necessary to handle the additional cache layer:

- `ADIOI_Sync_thread_start()`: is a new implemented routine providing cache synchronisation in ROMIO. It uses a dedicated POSIX thread to read data back from the cache file into the synchronisation buffer, and write it to the global file;
- `ADIOI_Cache_alloc()`: is a new implemented routine providing cache space allocation in ROMIO. It uses the `fallocate()` system call to efficiently allocate space in the local file system[2];
- `ADIOI_GEN_OpenColl()`: is the routine providing collective file open in ROMIO. In the new implementation, when `e10_cache` is set to `enable`, this routine also opens the cache file and stores its MPI file handle in the `cache_fd` field, added for the purpose inside the global MPI file handle. If for any reason the open of the cache file fails, the implementation reverts to standard open;
- `ADIOI_GEN_WriteContig()`: is the routine providing contiguous file write in ROMIO. In the new implementation, when `e10_cache` is set to `enable`, this routine uses the `cache_fd` file handle to write data. Additionally, it creates a synchronisation request (with associated `MPI_Request` handle) for the written extent [11] and sends it to `ADIOI_Sync_thread_start()`, which will take care of moving data to the global file system. When data transfer is complete the sync function invokes `MPI_Grequest_complete()` on the `MPI_Request` handle associated with the request;
- `ADIO_Close()`: is the routine providing file close in ROMIO. In the new implementation, when `e10_cache` is set to `enable`, this routine also invokes the `ADIOI_GEN_Flush()` routine to make

[2]For file systems that do not support the fallocate syscall the implementation reverts to standard allocation methods which physically writes zeros to the file, at the cost of time efficiency.

sure that all the data in the cache has been moved to the global file system, and finally closes the cache file as well as the global file;

- `ADIOI_GEN_Flush()`: is the routine providing file flushing in ROMIO. In the new implementation, when `e10_cache_flush_flag` is set to `flush_immediate`, it takes care of checking that previously created synchronisation requests have been completed by invoking `MPI_Wait()` on the associated `MPI_Request` handle. Alternatively, when `e10_cache_flush_flag` is set to `flush_onclose`, it sends all the pending synchronisation requests to `ADIOI_Sync_thread_start()` and then waits for them to complete as described above.

B. *Consistency Semantics*

As far as write consistency is concerned, the MPI-IO interface does not make any assumption regarding the underlying storage system or its semantics. ROMIO specifically supports file systems that are both POSIX compliant, like Lustre, and non-POSIX compliant, like NFS or PVFS. In MPI-IO, written data becomes globally visible only after either `MPI_File_sync()` or `MPI_File_close()` are invoked on the MPI file handle and by default there is no write atomicity. The motivation is that data can be cached at some level locally in the compute nodes. The ROMIO implementation can overcome the risk of data inconsistency, e.g. related to false sharing of file system blocks, using persistent file realms [15], and can even enforce atomicity using `MPI_File_set_atomicity()`.

In our implementation we comply to the MPI-IO semantics just described. This means that data written to the local file system cache using the newly introduced MPI-IO hints will be globally visible to the rest of the nodes only under the following circumstances:

- The `e10_cache_flush_flag` has been set to `flush_immediate` by the user and synchronisation, started automatically by the implementation right after the write operation, has completed;
- The `e10_cache_flush_flag` has been set to `flush_onclose` by the user and the invoked `MPI_File_close()` has returned;
- The `MPI_File_sync()` function has been invoked by the user and it has returned.

Consistency for reading data from the cache is not clearly defined by the ext2ph algorithm. In general, data written to the local file system cache can be read back from the user without accessing the global file system. Nevertheless, the algorithm calculates the location of a data block based on the number of aggregators, their logical position within the set of aggregators, and the size of the complete data set. This means that a collective read that matches the previous write could safely read the data from the aggregators' cache without incurring any problem. In spite of that, in general reading from the cache requires additional metadata describing the file layout across the caches. For this reason, we currently do not support reads from the local file system cache.

Furthermore, whenever required, we can enforce cache coherency ensuring that read operations cannot access data that is currently in transit, i.e., not or only partially moved from the cache to the global file. This can be done by locking the file domain extent being cached until all the data has been made persistent in the global file. For this purpose ROMIO provides a set of internal locking macros, namely `ADIOI_WRITE_LOCK`, `ADIOI_READ_LOCK` and `ADIOI_UNLOCK` that we used in our implementation. The lock of cached data can be selected by setting the `e10_cache` hint in Table II to `coherent`. This will `enable` the cache and set locks appropriately, assuming underlying file system support.

C. *Changes to the Application's Workflow*

Simplifying, most HPC applications can be divided into multiple phases of computation, in which data is produced, and I/O, in which data is written to persistent storage for post-processing purposes as well as defensive checkpoint-restart. Focusing on the I/O phase and considering the case of applications writing to a shared file, the I/O phase can be divided into the following steps:

1) The file is opened using `MPI_File_open()`: at this point the info object containing the user defined MPI-IO hints is passed to the underlying ROMIO layers.
2) Data is written to the file using `MPI_File_write_all()`: these functions invoke the underlying `ADIOI_GEN_WriteStridedColl()` previously described in Figure 2.
3) The file is closed using `MPI_File_close()`: after the file is closed data must be visible to every process in the cluster.

To take advantage of the proposed MPI-IO hint extensions, the application's workflow has to be modified. Figure 3 shows the classical application's workflow (cache disabled) as well as the modified version using the new hints (cache enabled). The difference is that, in order to take advantage of the proposed hints and hide the cache synchronisation to the computation phase, the `MPI_File_close()` for the I/O phase 'k' has been moved at the beginning of the I/O phase 'k+1', just before the new file is opened.

Since the workflow modification just presented might not be feasible for legacy applications, we developed a MPI-IO wrapper library (called MPIWRAP), written in C++, that can reproduce this change behind the scenes. The library can be linked to the application or preloaded with `LD_PRELOAD` and has been used for all the experiments contained in this paper. MPI-IO hints are defined in a configuration file and passed by the library to `MPI_File_open()`. We can define multiple hints targeting different files or groups of files. The library overloads `MPI_{Init,Finalize}()` and `MPI_File_{open,close}()` using the PMPI profiling interface. The workflow modification can be triggered for a specific set of files (identified by the same base name) in the configuration file. Whenever one of such files is closed, our `MPI_File_close()` implementation will return success. Nevertheless, the file will not be really closed.

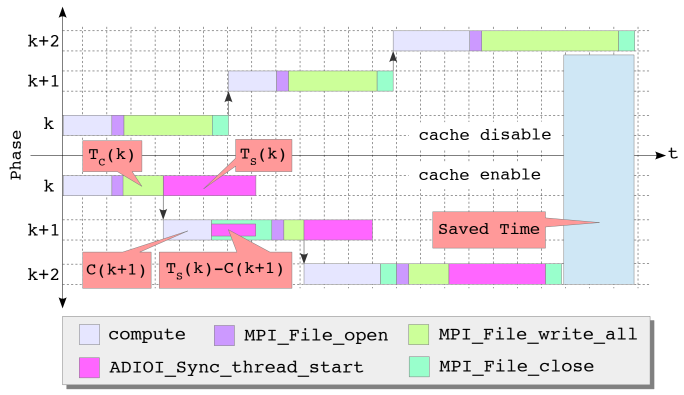

Fig. 3: Standard and modified workflows. When cache is disabled compute phase 'k+1' starts after file 'k' has been closed. When the cache is enabled compute 'k+1' can start immediately after data has been written. At the same time, background synchronisation of cached data starts. File 'k' is closed before the file 'k+1' is opened, forcing the implementation to wait for cache synchronisation to complete.

Instead, its handle will be kept internally for future references. When the next shared file with the same base name is opened, our `MPI_File_open()` implementation will search for outstanding opened file handles and will invoke `PMPI_File_close()` on them before opening the new file, thus triggering the cache synchronisation completion check for each of them.

D. I/O Bandwidth

According to the new I/O workflow, described in this section, we have that being $S(k)$ the amount of data written during phase 'k', $T_c(k)$ the time needed to write $S(k)$ collectively to the cache using `MPI_File_write_all()`, $T_s(k)$ the time needed to synchronise the cached data in every aggregator to the global file system (through `ADIOI_Sync_thread_start()`), and $C(k+1)$ the computation time of phase 'k+1', the resulting I/O bandwidth for 'k' is expressed by Equation 1:

$$bw(k) = \frac{S(k)}{T_c(k) + max(0,\ T_s(k) - C(k+1))} \tag{1}$$

Therefore, the total average bandwidth perceived by the application is:

$$BW = \frac{\sum_{k=0}^{N-1} S(k)}{\sum_{k=0}^{N-1} T_c(k) + max(0,\ T_s(k) - C(k+1))} \tag{2}$$

From Equation 1 (in which we have considered the open time neglectable) it is clear that the maximum performance can be obtained when $C(k+1) \geq T_s(k)$, that is, when we can completely hide cache synchronisation by the computation phase. On the other hand when $C(k+1) < T_s(k)$ we might have a minima in the bandwidth since `MPI_File_close()` needs to wait for cache synchronisation completion (Figure 3).

IV. Evaluation

To evaluate the proposed MPI-IO hints we use three popular I/O benchmarks frequently adopted to profile collective I/O performance in other research works: coll_perf[3], Flash-IO and IOR. Minor changes to the source code of the three benchmarks have been made to adapt them to our needs. For example, coll_perf and Flash-IO do not support writing to multiple files and the emulation of computing delays. Thus, we modified them to reproduce the workflow shown in Figure 3. The number of written files and a compute delay are now parameters that can be passed from the command line. In all our tests we used 512 MPI processes distributed over 64 nodes (8 procs/node), fixed the file stripe size to 4 MB and the stripe count to 4. Moreover, for simplicity, we also fixed the size of the cache synchronisation buffer (i.e. `ind_wr_buffer_size`) to 512 KB, which corresponds to the standard independent I/O buffer size. On the other hand, we varied the collective I/O parameters, i.e., the number of aggregators (from 8 to 64) and the collective buffer size (from 4 MB to 64 MB). For every combination of these parameters (<aggregators>_<coll_bufsize>) each benchmark writes four files of the same size (32 GB) with a compute delay of 30 seconds, which is in most cases enough to hide the synchronisation time. We compute the bandwidth as the average bandwidth over the four collective write operations (Equation 2). The different contributions within the collective I/O write path shown in Figure 2 are extracted from the ROMIO layer using MPE profiling [8]. Whenever the compute delay is not enough to hide synchronisation (e.g. when a small number of aggregators is used), the remaining synchronisation time is added to the total write time, thus reducing the bandwidth.

A. Testbed

Our testbed is a research cluster designed and developed in the context of the DEEP/-ER [2][3] projects (Dynamic Exascale Entry Platform/-Extended Reach). The DEEP/-ER cluster has 2048 cores distributed over 128 computing nodes (dual socket Sandy Bridge architecture). The storage system is composed of 6 Dell PowerEdge R520 servers equipped with 2 Intel Xeon Quad-core CPUs and 32 GB of memory and run the BeeGFS file system from Fraunhofer ITWM [7] (formerly known as FhGFS). The servers are connected to a SGI JBOD with 45 2TB SAS drives through a SAS switch using two 4x ports at 6 GB/s, for a total of four 8+2 RAID6 storage targets and 2 RAID1 targets for metadata and management data (1 drive is left as spare). One of the six I/O servers is dedicated as metadata server, one as management server and the remaining four as data I/O servers. Additionally, every compute node is equipped with 32 GB of RAM memory and a 80 GB SATA SSD containing the operating system plus an additional 30 GB ext4 partition (mounted under '/scratch') for general purpose storage. This partition, in our case, is used to locally cache collective writes. Finally, all the computing nodes are connected through an Infiniband QDR network and use ParaStation MPI [9] (PSMPI) version 5.1.0-1 as message passing library.

[3]Collective I/O benchmark distributed with the MPICH package.

B. *Coll_perf*

In our coll_perf configuration every process writes one 64 MB block being part of a tridimensional distributed array to a shared file, thus generating a strided pattern. For every experiment, in which we vary the number of aggregators and the collective buffer size, we measure the coll_perf perceived write bandwidth in three different cases: *1*) when writing directly to the global file system (BW Cache Disabled), *2*) when writing to the cache (BW Cache Enabled) and afterwards flushing its content to the global file system asynchronously, and *3*) when writing to the cache without flushing its content to the global file system (TBW Cache Enable). The last case provides the theoretical bandwidth achievable when the cache synchronisation cost is completely hidden. Additionally, our coll_perf experiments do not include the last write phase contribution (Figure 3). In fact, for the last write the cache synchronisation cost cannot be hidden since there is no following compute phase. We will show the effect that the last write has on the average bandwidth of the IOR benchmark at the end of this evaluation.

Figure 4 shows the write bandwidth for the three cases previously discussed. First of all, we can observe that for most of the experiments the aggregate bandwidth when using the cache is higher than the bandwidth measured when using only the global file system. In particular, we can reach a peak performance of about 20 GB/s, compared to the 2 GB/s of the standard case (BW Cache Disabled), which gives a ten fold improvement. Second, when the number of aggregators is equal to 8, we notice a reduced performance, as the synchronisation cost cannot be completely hidden.

The effect of the non-hidden cache synchronisation (not_hidden_sync) is shown in Figure 5. This figure presents the collective I/O performance breakdown for all the components shown in Figure 2. As we can see, although the theoretical bandwidth (TBW Cache Enable) peaks at 4 GB/s (Figure 4), the measured bandwidth (BW Cache Enable) can be even worse than the standard case (BW Cache Disable). This happens because the cache data cannot be flushed to the global file system quickly enough. In all the other experiments, the perceived and theoretical bandwidth are well aligned.

As already said in the previous sections, our SSDs based approach can also help to reduce the global synchronisation cost in the extended two phase I/O algorithm at the base of collective I/O. In fact, by comparing Figures 5 and 6, we see that the global synchronisation costs in collective I/O, represented by `MPI_Alltoall()` (shuffle_all2all) and `MPI_Allreduce()` (post_write) are consistently reduced when using the cache.

Finally, we observe that most of the times, when using the cache, larger collective buffers do not produce large performance improvements. This means that we can achieve good performance with small buffers and thus reduce the memory pressure of collective write operations on compute nodes.

C. *Flash-IO*

Flash-IO is the I/O kernel of the Flash application. Flash is a block-structured adaptive mesh hydrodynamics code. The computational domain is divided into blocks which are distributed across the processors. Typically a block contains 16 zones in each coordinate direction (x,y,z) and a perimeter of guard cells (presently 4 zones deep) to hold information from the neighbors. The application writes three files using the parallel HDF5 library: a checkpoint file, a plot file without corner data and a plot file with corner data. The checkpoint file is the biggest of the three and consumes the majority of the I/O time. In our configuration the checkpoint file contains 80 blocks/process and each of the 16 zones/block contains 24 variables encoded with 8bytes (768 KB/proc/block). Therefore, the total size is slightly bigger than 30 GB (including metadata).

Figure 7 shows the write bandwidth perceived by Flash-IO for all the experiments performed in the different cases under study. Similarly to coll_perf, when the cache is enabled we can hide the cache synchronisation cost for most of the experiments. Once again, like in the previous case, when the number of aggregators is equal to 8 we have a mismatch

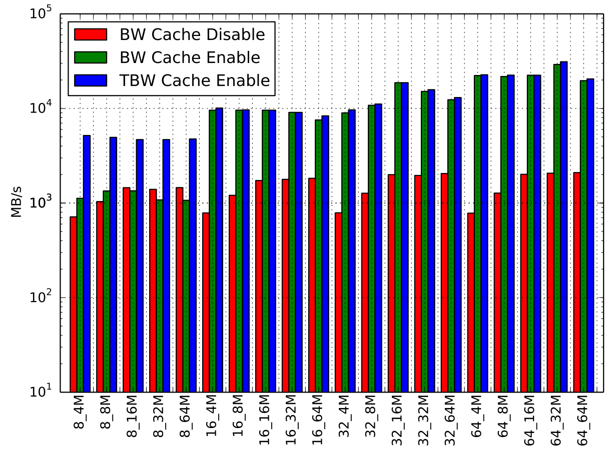

Fig. 4: coll_perf perceived bandwidth for all combinations of <aggregators>_<coll_bufsize>.

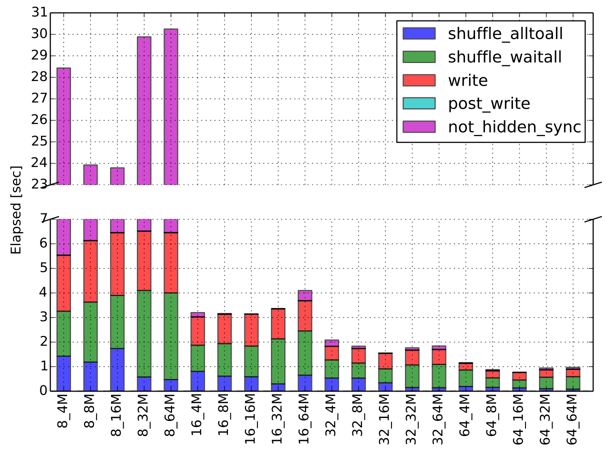

Fig. 5: coll_perf collective I/O contribution breakdown when cache is enabled.

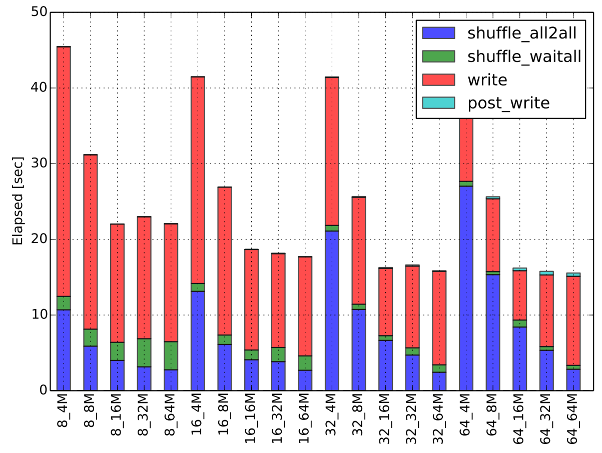

Fig. 6: coll_perf collective I/O contribution breakdown when cache is disabled.

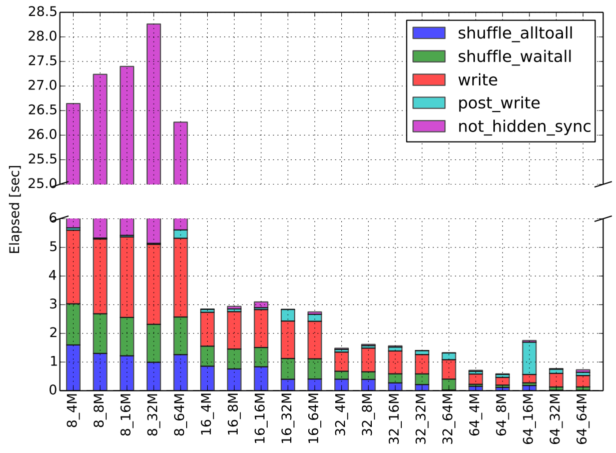

Fig. 8: Flash-IO collective I/O contribution breakdown when cache is enabled.

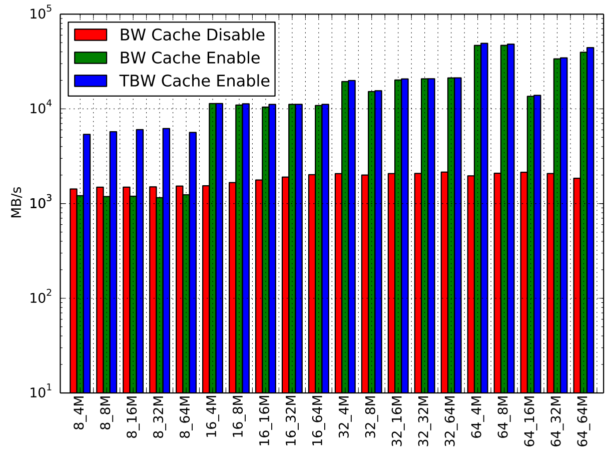

Fig. 7: Flash-IO perceived bandwidth for all combinations of <aggregators>_<coll_bufsize>.

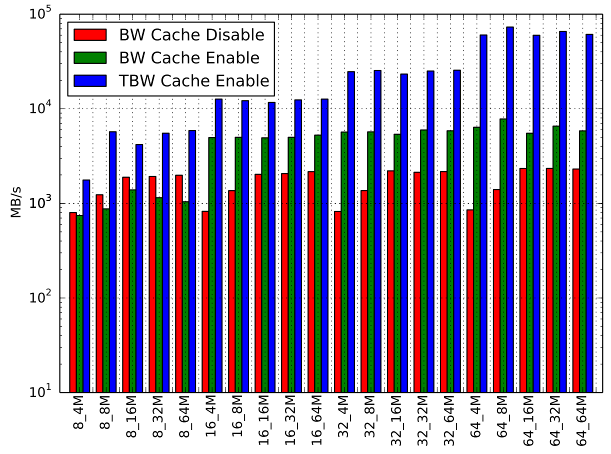

Fig. 9: IOR perceived bandwidth for all combinations of <aggregators>_<coll_bufsize>.

between the perceived and the theoretical bandwidth. For Flash-IO the peak bandwidth is about 40 GB/s when using 64 aggregators and 4 MB collective buffer size, against the 2 GB/s measured when writing directly to the parallel file system.

Figure 8 shows the effect of cache usage on the different collective I/O performance contributions. We can clearly see that when the number of aggregators is equal to 8 the cache synchronisation cannot be completely hidden, causing the bandwidth mismatch previously observed in Figure 7. Furthermore, like in the coll_perf case the global synchronisation contributions can be reduced when using the cache, and so can the memory pressure on the compute nodes. Nevertheless, for the 64 aggregators and 16 MB collective buffer size configuration the global synchronisation overhead associated to `MPI_Allreduce()` (post_write) has a larger value. The outlier causes a strong reduction in the write bandwidth, although we can still achieve more than 10 GB/s. This indicates that the effect of global synchronisation when using the cache can be even more severe, due to the much higher bandwidth achievable compared to the standard global file system approach.

D. IOR

We tested IOR when writing collectively to a shared file. Each of the 512 MPI processes writes one 8 MB block of data for each of the 8 segments, that is, a 32 GB file during every test. As for the previous two benchmarks, Figure 9 shows the write bandwidth perceived by IOR. Nevertheless, unlike the previous benchmarks, in IOR we also take into account the non-hidden synchronisation cost coming from the last write phase. In this case, the peak bandwidth when using the cache can reach about 6 GB/s versus the 2 GB/s of the standard collective write to the global file system. We can also see that the theoretical bandwidth is aligned with the figures presented for coll_perf and Flash-IO. In fact, although we can hide cache synchronisation costs for the three intermediate write phases in IOR, the fourth write phase will limit the peak performance.

Figures 10 shows the collective I/O cost breakdown for all the time contributions. In this figure we can clearly observe the

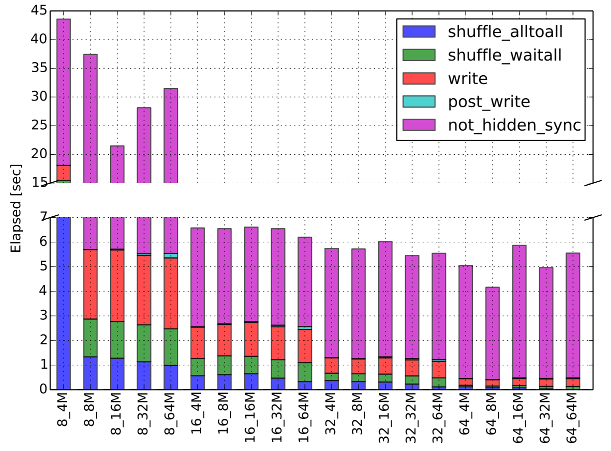

Fig. 10: IOR collective I/O contribution breakdown when cache is enabled.

not_hidden_sync term preventing IOR from achieving higher performance. This is added to the other time contributions and corresponds to the term $T_s(k) - C(k+1)$ reported in Equation 1. In our specific case, $k = 4$ and $C(4+1) = 0$, meaning that the total write time is accounting for the whole $T_s(4)$ term.

V. Related Work

Many research works have tried to optimise collective I/O focusing on different aspects. Yu and Vetter [30] before us have identified the global synchronisation problem as one of the most severe for collective I/O performance. They exploited access pattern characteristics, common in certain scientific workloads, to partition collective I/O into smaller communication groups and synchronise only within these. Block-tridiagonal patterns, not directly exploitable, are automatically reduced, through an intermediate file view, to a more manageable pattern and can thus take advantage of the proposed solution. The ADIOS library [22] addresses this problem similarly by dividing a single big file into multiple files to which collective I/O is carried out independently for separated smaller groups of processes. Lu, Chen, Thakur and Zhuang [23] further explored collective I/O performance beyond global synchronisation and considered memory pressure of collective I/O buffers. They proposed a memory conscious implementation that accounts for reduced memory per core in future large scale systems. Liao [19] focused on the file domain partitioning impact on parallel file systems' performance. He demonstrated that by choosing the right file domain partitioning strategy, matching the file system locking protocol, collective write performance can be greatly improved. Yong, Xian-He, Thakur, Roth and Gropp [13] addressed the problem of I/O server contention using a layout aware strategy to reorganize data in aggregators. On the same lines, Xuechen, Jiang and Davis [31] proposed a strategy to make collective I/O 'resonant' by matching memory layout and physical placement of data in I/O servers and exploiting non-contiguous access primitives of PVFS2. The strategy proposed is similar in concept to the Lustre implementation of collective I/O in which file contiguous patterns are converted to stripe contiguous patterns and the concurrency level on OSTs can be set using the MPI-IO hint `romio_lustre_co_ratio` (Client-OST ratio). Liu, Chen and Zhuang [21] exploited the scheduling capabilities of PVFS2 I/O servers to rearrange I/O requests' order and better overlap read and shuffle phases among different processes.

Lee, Ross, Thakur, Xiaosong and Winslett [18] proposed RTS as infrastructure for remote file access and staging using MPI-IO. Similarly to our approach, RTS uses additional threads, Active Buffering Threads (ABT) [24], to transfer data in background to the compute phase. Moreover, the authors also modified the ABT ROMIO driver implementation to stage data in the local file system whenever the amount of main memory runs low. Although they include collective I/O in their study, they lack a detailed evaluation of the impact that SSD caching can have on the different performance contributions of collective I/O and the additional reduction of memory pressure. Furthermore, remote staging of data requires additional nodes while we collocate storage with compute. The SCR library [10] also uses local storage resources to efficiently write checkpoint/restart data but this is targeted to a specific use case and requires the modification of the application's source code to be integrated. Other works, focus on I/O jitter reduction using multi-threading and local buffering resources [16], but we do an evaluation of collective I/O and show how the effect of I/O jitter can become even more prominent when using fast NVM devices. More recently the Fast Forward I/O project [6], from U.S. Department of Energy (DOE), proposed a burst buffer architecture to absorb I/O bursts from file system clients into a small number of high performance storage proxies equipped with high-end solid state drives. This technique has been, e.g., implemented in the DDN Infinite Memory Engine [1]. Even though the burst buffer solution is interesting, it may require very expensive dedicated servers as well as significant changes to the storage system architecture.

Unlike previous works, we proposed a fully integrated, prototype solution for new available memory technologies able to scale aggregate bandwidth in collective I/O with the number of available compute nodes. Additionally, our solution does not require any proprietary hardware or dedicated kit to work. We demonstrate that SSD based cache can reduce the synchronisation overhead intrinsic in the collective I/O implementation in ROMIO as well as the requirement for large collective buffers (memory pressure). Our implementation is compatible with legacy codes, since it does not require any change at the application level, and can work out of the box with any backend file system, although in DEEP-ER we focused on BeeGFS. At the moment the cache synchronisation is implemented in the ADIO UFS driver using pthreads. Future releases of BeeGFS will support native caching, including asynchronous flushing of local files to global file system. We have already integrated ROMIO with a BeeGFS driver that will take advantage of these functionalities.

VI. Conclusions and Future Work

In this paper we have presented a new approach in using multi-tier storage systems to improve collective I/O performance in ROMIO. We have demonstrated that the usage of SSDs attached to compute nodes can improve collective I/O by increasing the aggregated I/O bandwidth, reducing the global synchronisation overhead, and finally the memory requirements. This can be done provided there is enough computation to hide background cache synchronisation costs.

At the moment local SSDs are not fully integrated in the storage hierarchy and researchers will have to figure out how to best exploit them. Burst buffers provide a possible solution but these require expensive dedicated kit, whereas we can use inexpensive commodity flash/solid state devices in computing nodes. In the short term we plan to test our SSD based approach with real applications from the DEEP-ER project, further validating the introduced benefits in real world workloads. In the longer term we plan to support cache reading operations and in general more complex and better integrated caching support and policies. In this direction, the Exascale10 initiative plans to build an Exascale ready middleware that will be able to efficiently support multiple integrated storage tiers in the I/O stack.

Acknowledgment

This work has been funded by the FP7 program of the European Commission through the DEEP-ER project (Grant Agreement no. 610476) and supported by the Exascale1O (E10) initiative.

References

[1] DDN Infinite Memory Engine. http://www.ddn.com/products/infinite-memory-engine-ime14k/.

[2] DEEP, Dynamic Exascale Entry Platform. http://www.deep-project.eu.

[3] DEEP-ER, Dynamic Exascale Entry Platform - Extended Reach. http://www.deep-er.eu.

[4] Exascale10 (E10) initiative. http://www.exascale10.com.

[5] Exascale10 in DEEP-ER public repository. https://github.com/gcongiu/E10.git.

[6] Fast Forward I/O project. https://users.soe.ucsc.edu/~ivo/blog/2013/04/07/the-ff-stack.

[7] Fraunhofer File System. http://www.beegfs.com/cms.

[8] MPI Parallel Environment. http://www.mcs.anl.gov/research/projects/perfvis/software/MPE.

[9] Parastation MPI. http://www.par-tec.com/products/parastation-mpi.html.

[10] Scalable Checkpoint Restart library. http://computation.llnl.gov/projects/scalable-checkpoint-restart-for-mpi.

[11] "MPI: a message-passing interface standard," Report, pp. 473–482, 2012. [Online]. Available: http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

[12] P. H. Carns, W. B. Ligon, III, R. B. Ross, and R. Thakur, "Pvfs: A parallel file system for linux clusters," in *Proceedings of the 4th Annual Linux Showcase and Conference*. Atlanta, Georgia, USA: USENIX Association, Oct. 2000, pp. 317–327.

[13] Y. Chen, X.-H. Sun, R. Thakur, P. Roth, and W. Gropp, "Lacio: A new collective i/o strategy for parallel i/o systems," in *Proceedings of the 25^{th} IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Anchorage, Alaska, USA, May 2011, pp. 794–804.

[14] A. Ching, A. N. Choudhary, W. Liao, L. Ward, and N. Pundit, "Evaluating i/o characteristics and methods for storing structured scientific data," in *Proceedings of the 20^{th} IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Rhodes Island, Greece, Apr. 2006.

[15] K. Coloma, A. Choudhary, W.-K. Liao, L. Ward, E. Russell, and N. Pundit, "Scalable high-level caching for parallel i/o," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, April 2004, pp. 96–.

[16] M. Dorier, G. Antoniu, F. Cappello, M. Snir, and L. Orf, "Damaris: How to efficiently leverage multicore parallelism to achieve scalable, jitter-free i/o," in *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, Sept 2012, pp. 155–163.

[17] J. He, H. Song, X.-H. Sun, Y. Yin, and R. Thakur, "Pattern-aware file reorganization in mpi-io," in *Proceedings of the 6^{th} Parallel Data Storage Workshop (PDSW)*, Seattle, Washington, USA, 2011, pp. 43–48.

[18] J. Lee, R. Ross, R. Thakur, X. Ma, and M. Winslett, "Rfs: efficient and flexible remote file access for mpi-io," in *Cluster Computing, 2004 IEEE International Conference on*, Sept 2004, pp. 71–81.

[19] W. Liao, "Design and evaluation of MPI file domain partitioning methods under extent-based file locking protocol," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 2, pp. 260–272, 2011.

[20] W. Liao and A. N. 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 (SC)*, Austin, Texas, USA, Nov. 2008.

[21] J. Liu, Y. Chen, and Y. Zhuang, "Hierarchical I/O scheduling for collective I/O," in *Proceedings of the 13^{th} IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid)*, Delft, Netherlands, May 2013, pp. 211–218.

[22] Q. Liu, J. Logan, Y. Tian, H. Abbasi, N. Podhorszki, J. Y. Choi, S. Klasky, R. Tchoua, J. Lofstead, R. Oldfield, M. Parashar, N. Samatova, K. Schwan, A. Shoshani, M. Wolf, K. Wu, and W. Yu, "Hello adios: the challenges and lessons of developing leadership class i/o frameworks," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 7, pp. 1453–1473, 2014. [Online]. Available: http://dx.doi.org/10.1002/cpe.3125

[23] Y. Lu, Y. Chen, R. Thakur, and Y. Zhuang, "Abstract: Memory-conscious collective I/O for extreme-scale HPC systems," in *Proceedings of the 2012 High Performance Computing, Networking, Storage and Analytics (SCC)*, Salt Lake City, UT, USA, Nov. 2012, pp. 1360–1361.

[24] X. Ma, M. Winslett, J. Lee, and S. Yu, "Improving mpi-io output performance with active buffering plus threads," in *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, April 2003, pp. 10 pp.–.

[25] J. Prost, R. Treumann, R. Hedges, A. E. Koniges, and A. White, "Towards a high-performance implementation of MPI-IO on top of GPFS," in *Proceedings of the 6^{th} International Euro-Par Conference (Euro-Par)*, Munich, Germany, Aug. 2000, pp. 1253–1262.

[26] R. Thakur and A. N. Choudhary, "An extended two-phase method for accessing sections of out-of-core arrays," *Scientific Programming*, vol. 5, no. 4, pp. 301–317, 1996.

[27] R. Thakur, W. Gropp, and E. Lusk, "An abstract-device interface for implementing portable parallel-i/o interfaces," in *Proceedings of the 6^{th} IEEE International Symposium on Frontiers of Massively Parallel Computation (FRONTIERS)*. IEEE Computer Society Press, 1996, pp. 180–187.

[28] ——, "Data sieving and collective i/o in romio," in *Proceedings of the 7th IEEE International Symposium on Frontiers of Massively Parallel Computation (FRONTIERS)*, 1999, p. 182.

[29] L. Ying, "Lustre ADIO collective write driver," White Paper, October 2008.

[30] W. Yu and J. Vetter, "Parcoll: Partitioned collective i/o on the cray xt," in *Proceedings of the 37^{th} International Conference on Parallel Processing (ICPP)*, Sep. 2008, pp. 562–569.

[31] X. Zhang, S. Jiang, and K. Davis, "Making resonance a common case: A high-performance implementation of collective i/o on parallel file systems," in *Proceedings of the 2009 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2009, pp. 1–12.