Data Redistribution using One-sided Transfers to In-memory HDF5 Files

Jerome Soumagne[1,2], John Biddiscombe[1], and Aurélien Esnard[2]

[1] Swiss National Supercomputing Centre
Galleria 2, Via Cantonale, 6928 Manno, Switzerland
[2] INRIA Bordeaux Sud-Ouest
351 cours de la Liberation, 33405 Talence, France

Abstract. Outputs of simulation codes making use of the HDF5 file format are usually and mainly composed of several different attributes and datasets, storing either lightweight pieces of information or containing heavy parts of data. These objects, when written or read through the HDF5 layer, create metadata and data IO operations of different block sizes, which depend on the precision and dimension of the arrays that are being manipulated. By making use of simple block redistribution strategies, we present in this paper a case study showing HDF5 IO performance improvements for "in-memory" files stored in a distributed shared memory buffer using one-sided communications through the HDF5 API.

Keywords: Data Redistribution, Distributed Shared Memory, HDF5, One-sided Communication

1 Introduction

HDF5 [11], the Hierarchical Data Format, allows users to write data output in a very flexible manner. One file can be composed of different datasets, usually containing a large amount of data, and of attributes, storing small pieces of information. Datasets can be simple scalars or N-dimensional vectors written in parallel using hyperslab selections – these selections depend entirely on the code implementation. Parallel writes or reads can be issued in a uniform manner or can follow a totally random pattern. Concurrent with these data IO operations, HDF5 metadata is written and can be accessed several times if objects are opened, created or closed or if the metadata has not been previously cached. Therefore a complete HDF5 file write or read in parallel may consist of a large number of accesses in a complex pattern.

The HDF5 architecture allows the creation of customized IO methods called drivers, one well-known parallel driver is the MPI-IO driver, discussed in section 2. Disk IO being a significant and now commonplace bottleneck in simulations, we developed a parallel virtual file driver called the DSM driver which allows one to redirect HDF5 IO operations in parallel to a distributed shared memory (DSM) buffer (the reader is referred to [9] and [8] for a more complete introduction to the DSM driver and communicators). Simulation processes may write

in-memory HDF5 files using various types of communication, the principal intended use of these in-memory files being code-coupling of parallel applications. The original implementation made use of two-sided communication only; we recently extended it to make use of one-sided communication – we focus in this paper only on one-sided transfers and consider the case where the nodes hosting the DSM are different from the nodes hosting the simulation processes, i.e. where traffic between them *must* traverse the network. We present in section 3 the MPI one-sided communicator used for this study, along with an additional communicator specially designed for the Cray XE6.

In the original implementation, HDF5 files are written using a linear address space where the file grows in size by extending upwardly the address range used. The addresses are spread (evenly by default) across a series of DSM host processes so that as the file grows in size and data is written into higher addresses more network links are utilized and the higher the transfer bandwidth *should* be. In practice, most data reads/writes for datasets or hyperslabs are significantly smaller than the entire file and thus use only a small number of memory partitions – and hence network links, at any given time, which limits the bandwidth reached. We extend this strategy in section 4 by remapping the address space nonlinearly using varying block sizes among DSM host processes (thereby distributing traffic more evenly). We present a case study showing the performance obtained in section 5 and compare it to related studies in section 6.

2 HDF5 File IOs

HDF5 IOs can be produced in very different ways. As mentioned above, *drivers* allow users to select a suitable IO mechanism for the system. One frequently used driver is the MPI-IO driver, best suited for parallel file systems, since it uses MPI-IO underneath. Whilst MPI-IO and implementations such as ROMIO [10] have been optimized for various types of accesses depending on the file system used, HDF5 also provides its own ways of tuning and writing data in parallel. For instance, the chunking mechanism allows files and particularly datasets to be stored in a non-contiguous form, i.e. in equally sized chunks, which can be helpful for parallel file systems, over which datasets can therefore be striped. Additional optimizations have also been made in the MPI-IO driver and HDF5 library itself for specific file systems such as the Lustre file system [6].

These enhancements are particularly useful in a traditional pipeline model where data is archived and post-processed from file systems, however bandwidth offered by file systems is limited. Introducing the DSM driver in the pipeline allows us to couple two different applications in parallel through the network by using the HDF5 interface. This offers an additional exchange method before saving post-processed data to disk for archiving purposes. Parallel optimizations implemented in the HDF5 library can be re-used by the DSM driver, such as the chunking mechanism, but other types of accesses specific to file systems need to be adapted and re-optimized within the driver itself.

3 DSM Driver and Communicators

As opposed to the MPI-IO driver, where the application is effectively *coupled* to the file-system, when using the DSM driver, two applications – parallel simulation and DSM host (integrating post processing code) – are coupled together through a communication layer, referred to as an inter-communicator. The DSM architecture being modular, permits different inter-communicator types to be implemented, which can follow one-sided or two-sided communication patterns. For this case study, two different one-sided inter-communicators are considered: one based on MPI RMA and one specific to Cray systems, based on an API called DMAPP.

MPI RMA Inter-communicator. The MPI RMA communicator makes use of the passive MPI RMA communication mechanism [5]. When the DSM is allocated, `MPI_Alloc_mem` is called and the window is defined as the size of the requested HDF5 file. `MPI_Put` can then be issued in a one sided manner using `MPI_Win_lock` and `MPI_Win_unlock` between transactions.

The communicator can be dynamically created (using the dynamic process management set of functions) but due to the numerous restrictions imposed by MPI implementations, on large systems (e.g. on Cray systems), the communicator has to be defined using an `MPI_Intercomm_create` call within an MPMD job (where the global communicator has been previously split between applications).

DMAPP Inter-communicator. The DMAPP communicator is derived from the aforementioned MPI RMA communicator. On Cray machines that support the latest generation of interconnect, Gemini [3], Cray defines the Distributed Memory Application API, referred as DMAPP [4]. This API is used on these systems to implement one-sided libraries such as Cray SHMEM and is also used by PGAS compilers (Co-array Fortran and UPC). We have implemented a communicator taking direct advantage of this lower level one-sided communication library. On the simulation side, to avoid memory overheads created by symmetric memory usage, we make use of non-symmetric memory, allocated and registered to the DMAPP API on the DSM hosts only. This registration step provides memory segment information which is then exchanged with the simulation (only once at initialization time, assuming that the DSM size is fixed between time steps). `dmapp_put` calls can then be issued to transfer data into the DSM.

4 Redistribution Strategies

In our implementation the DSM is distributed among p processes, each process allocating l bytes of data, which gives a total DSM length of $L = l \times p$. Using linear addressing, the DSM is contiguously filled from process rank 0 to process rank $(p-1)$. If a simulation writes a file of size S, the actual number of processes used to receive data will thus be $\lceil \frac{S}{l} \rceil$ with $S \leq L$. Whilst this method can provide

relatively good performance when $S \simeq L$, if the file written is composed of several different datasets (i.e. each much smaller than L), which are contiguously (and sequentially) mapped onto the DSM, individual simulation processes will waste bandwidth by using only a small partition of the network links available – particularly so when datasets are divided between simulation processes and written using hyperslab selections. We therefore sought better strategies which could be enabled on demand.

4.1 Mask Redistribution

When $S \ll L$, a first simple strategy is to automatically re-size the DSM window to the requested file size without any concrete memory reallocation. This can effectively improve the overall bandwidth by making $S \simeq L$ but this brings two main drawbacks: the most evident one is that it wastes memory allocated on the DSM, the second one is that it does not solve the multiple dataset problem mentioned above.

4.2 Block Cyclic Redistribution

The second strategy to be considered in this case study is a block cyclic redistribution [13]. It is a simple strategy and it potentially allows a good load balance between DSM processes. A block size s being fixed, the DSM address mapping is decomposed into $\frac{L}{s}$ blocks. For convenience, the DSM length L is adapted so that it becomes a multiple of s. Blocks are distributed in a round-robin fashion, the B^{th} block is sent to the process rank ($B \bmod P$) or ($B \bmod B_c$) (if B_c, the number of blocks in a cycle is not equal to P, the number of processes). Hence every address a is associated to the following triplet (p, o, i) which can be written as:

$$a \mapsto \left(B \bmod P, \left\lfloor \frac{B}{P} \right\rfloor , a \bmod s \right) \tag{1}$$

the first term p being the process index within the DSM, o the local block offset in a process and i the local address offset within a block.

This method presents two obvious advantages: bandwidth is not wasted even if $S \ll L$; data chunks are load balanced, which is especially beneficial when multiple datasets are written. However this method can potentially create a huge number of data transactions, depending on the block size chosen, which could result in a performance drop.

4.3 Random Block Redistribution

The third strategy tested consists in re-using the algorithm previously described, scattering the DSM address space into pieces of size s. Another step is then added to the redistribution pipeline, shuffling the blocks in a randomized but constant manner (so that blocks can be retrieved).

This method can present another advantage compared to the previous solution (but keeps the same main drawback), it may avoid a possible network

congestion if two simulation processes were sending data to the same DSM process using the block cyclic redistribution algorithm – which may occur with a periodic frequency introduced by certain communication patterns and data distributions in the file.

5 Performance Evaluation

For these tests, we use two systems: an InfiniBand QDR 4X cluster with MVAPICH2 [2] composed of 15 nodes (180 cores) and a Cray XE6 system composed of two racks, i.e. 176 compute nodes (4224 cores), with Cray MPT (derived from MPICH2 [1] [7]). To be able to evaluate the performance obtained using the previously defined strategies, we first run micro-benchmarks on these two machines.

5.1 Internode Micro-benchmark

The micro-benchmarks are derived from the OSU test suite [2] and identify the bandwidth performance on the different systems for different sizes of packets between two different nodes. Only put operations are tested here. Results are shown below in figure 1.

A careful examination of these charts shows a performance drop point with MVAPICH2 for packets of 16KB, though the overall bandwidth reflects InfiniBand QDR 4X performance. For the XE6, theoretical unidirectional performance is estimated at 5GB/s. Here the DMAPP interface performs better than the MPI one-sided interface. Two main drop points can however be noticed, 4KB for DMAPP and 1KB for MPI – these points correspond to the standard offload thresholds, making use of the RDMA engine for large messages.

5.2 Single Dataset Benchmark

For the following benchmarks, write bandwidth tests can be seen as basic client-server tests: a first set of processes (servers) hosts the DSM and waits for incoming data, a second set of processes (clients) writes HDF5 data in parallel to the DSM using the HDF5 DSM driver. The measured bandwidth corresponds to the average time of a complete file write (HDF5 create, write and close operations). The first benchmark writes one file composed of one single dataset using hyperslab selections.

Contiguous/Linear distribution. The DSM is distributed among 8 nodes (32 processes, 4 per node) on the InfiniBand cluster and among 88 nodes (176 processes, 2 per node) on the XE6. To keep a certain consistency between the systems, the local buffer size allocated per node is kept to 512MB, which creates a DSM of $8 \times 512 = 4$GB on the InfiniBand cluster and a DSM of $88 \times 512 = 44$GB on the XE6. Given this fixed DSM (file) size, a single dataset of the matching size

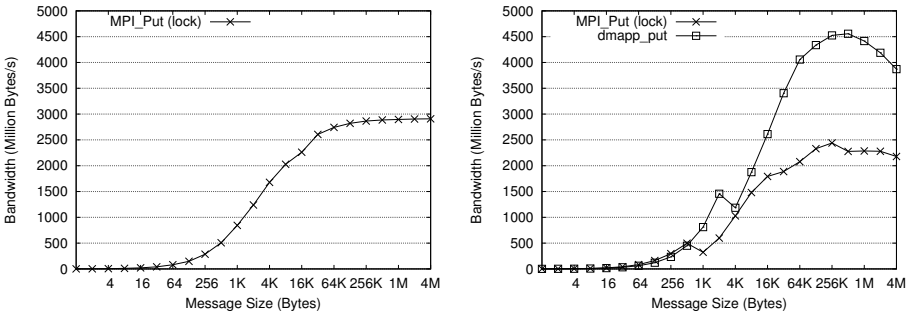


Fig. 1. Internode bandwidth micro-benchmark – (Left) InfiniBand QDR 4X cluster with MVAPICH2 – (Right) Cray XE6 with Cray MPT and DMAPP.

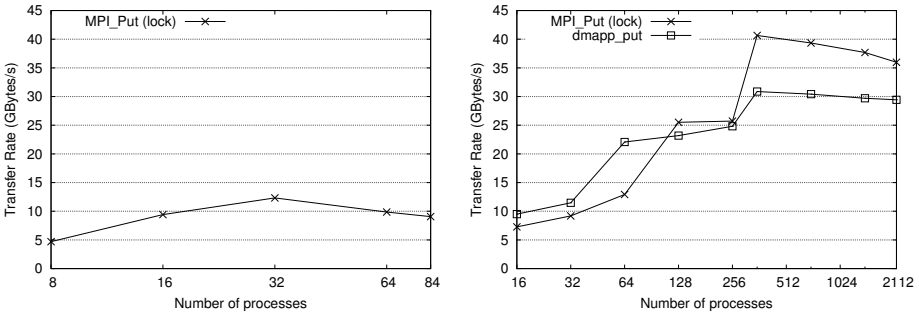


Fig. 2. Write transfer rate of an (in-memory) HDF5 file composed of one single dataset using contiguous distribution – (Left) InfiniBand QDR 4X cluster – (Right) Cray XE6.

is written from the combined send nodes (smaller pieces per process as number of processes increases).

For writing, on the XE6, the number of processes is 4 per send node until 88 nodes are reached (352 processes) at which point, processes per send node are increased up to 24 – giving 2112 processes writing data in total. On the Infiniband cluster, 7 send nodes are available and 4 processes per node are used initially and then incremented to 12 per send node giving a maximum of 84 send processes. (Note that 4 processes per send node were selected as the starting point, because with fewer processes injecting data, we are unable to fully utilize the individual network links). Therefore, as shown in figure 2: on the InfiniBand cluster, a peak bandwidth is observed at 12.5GB/s with 32 processes (8 receive and 7 send links active); on the XE6, at 40.5GB/s with 352 processes (88 send and receive links active). Note that the XE6 system used for the tests has a 2D torus ($1 \times 6 \times 16$), and the resulting bandwidth is lower than that achievable using a 3D torus.

Block Cyclic and Random Block Redistributions. For different block sizes, we run the same benchmark as above, using a single dataset. This test

allows us to evaluate block redistribution advantages as opposed to a simple contiguous distribution. Results are presented in figures 3 and 4.

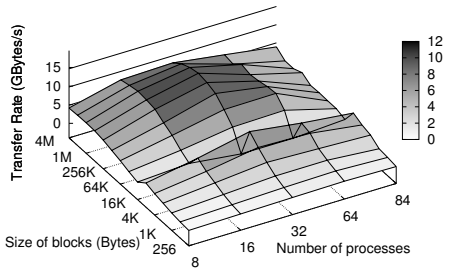
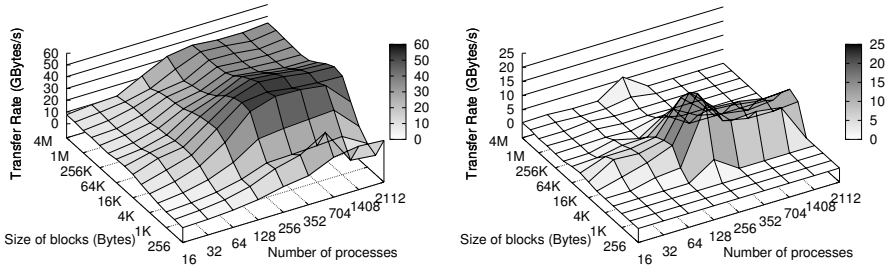
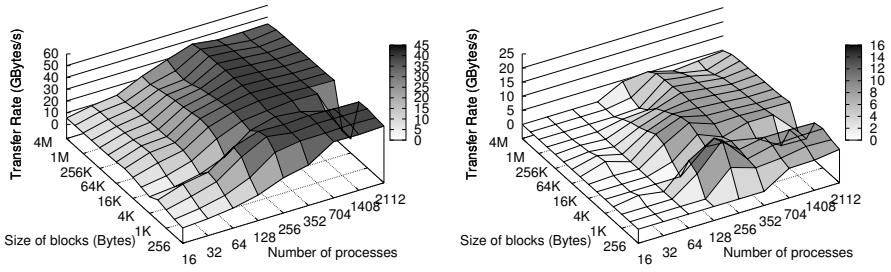


Fig. 3. Write transfer rate on InfiniBand QDR 4X cluster of an (in-memory) HDF5 file composed of one single dataset using block cyclic redistribution.



(a) MPI RMA communicator.



(b) DMAPP communicator.

Fig. 4. Write transfer rate on Cray XE6 of an (in-memory) HDF5 file composed of one single dataset – (Left) Block cyclic redistribution – (Right) Performance comparison (difference) between block cyclic and contiguous distributions.

On both systems, the bandwidth drop points of section 5.1 can be observed. While these drop points had a small effect on the micro-benchmark, they lead to a significant slow-down in the HDF5 write operations when those block sizes are used repeatedly. On the XE6 system, a significant improvement compared to the contiguous write is evident for block sizes belonging to the [16KB; 64KB] interval with the MPI RMA communicator and for block sizes below 4KB for the DMAPP communicator. Since metadata operations are usually very small transfers, being able to use this communicator in combination with the MPI RMA communicator is an advantage for this system. However one can also notice in figure 3 that there is no improvement at all on the InfiniBand cluster when using a block cyclic method if *only a single dataset* is written into the file (compare the peak transfer rate to that of the left of figure 2). The transfer rates for the full DSM sized dataset are in general slower using the block/random redistribution on the InfiniBand system and this is because breaking the data writes into many smaller blocks does not improve performance as can be seen from the micro-benchmark result of figure 1.

Random block results are not shown here – for brevity – but globally increase the bandwidth as one may see in the next benchmark, and avoid possible congestion issues in the DSM.

5.3 Multiple Dataset Benchmark

To reflect the behaviour of a common simulation code, the previous benchmark is reused here, this time creating a file composed of 10 datasets instead of a single one. The same configuration is used as the previous tests, each of the datasets has the same fixed size and their sum is the size of the allocated DSM, i.e. 4GB for the InfiniBand cluster and 44GB for the XE6. Results are shown in figure 5.

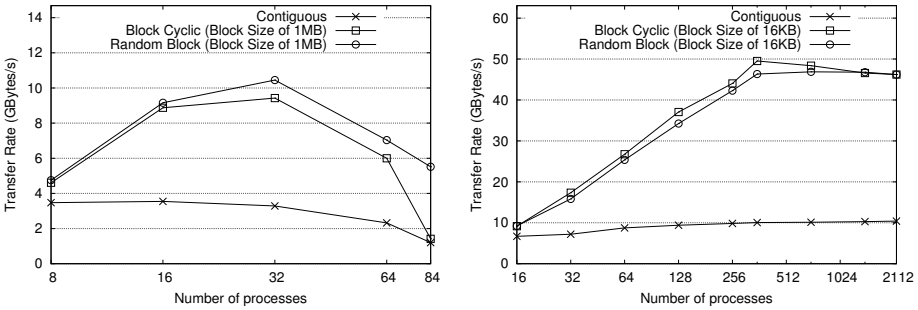


Fig. 5. Write transfer rate using MPI RMA communicator of an (in-memory) HDF5 file composed of 10 datasets – (Left) InfiniBand QDR 4X cluster – (Right) Cray XE6.

It is evident from this figure that writing using block redistribution is much more efficient than linear mapping. Since each dataset *in the linear HDF5 memory space* is contiguous, writing 10 datasets in parallel but sequentially in time,

causes only one tenth of the links to become active for each individual dataset. By redistributing blocks for each of the much smaller datasets across all processes, we make use of all of the links for all of the transfers. For block cyclic redistribution, providing each dataset is at least $s \times P$ in size, the data will be well distributed.

It is perhaps surprising that the graph of figure 5 (left) shows a significant drop in transfer rate as the number of send processes increases. The drop is smaller for random distribution than for cyclic and this can be explained by noting that we have used 4 processes per listening node, so in fact the cyclic redistribution hits the same link 4 times in succession, which will not generally happen for the random distribution. We therefore see a more gradual fall off in line with figure 2 (left) for the random mode, the overall drop being caused by the increase in individual number of transfers as the effect of latency and lower performance for smaller packets dominates.

6 Related Work and Discussion

The results presented here appear to be typical for the kinds of system tested, but can however be affected by the network topology and capabilities, system configuration, number of nodes used, number of processes per node, and so on. The space of potential combinations of parameters for plots is beyond what can be presented in a short paper so certain decisions as the number of processes to use per node were made to try to maximize the data injection and network saturation to give representative results. Note that the implementation of MPI RMA for the XE6 is not yet optimized and the measured performance for large messages (above 1KB) should be improved in the future [7]. Absolute bandwidths may not therefore be indicative of all installations – though this does not affect our results.

The improvements in transfer rates found when using redistribution are broadly in line with expectations. In fact the advantages of data redistribution are well known and date back to the origins of message passing [13]. Many projects have made use of block cyclic distribution as a means of improving performance for scattered data, in particular PGAS languages (such as UPC [12]) provide options for shared array allocation using block cyclic layouts, which can improve algorithmic performance. In fact our flexible communicator design opens up the possibility that a PGAS based layer could be used directly instead of MPI or DMAPP as we have presented here and we shall pursue this in future work.

The observation that certain packet sizes are handled better by different APIs also allows the possibility of further fine tuning transfers. IO operations from HDF5 applications typically consist of many small metadata and larger heavy data requests and these different needs can be served by switching communicators on the fly to use DMAPP for metadata and MPI RMA for heavy data decomposed into blocks.

7 Conclusion

We presented in this paper a case study where HDF5 files are sent to a DSM using one-sided transfers and found that implementing redistribution strategies significantly improves the performance of data writes for typical use cases where multiple datasets are written into a much larger file. By choosing block sizes that are optimal for the underlying hardware and matching the number of send/receive nodes, we are able to improve the data bandwidth. Codes coupled using the DSM driver are now able to communicate at speeds approaching the maximum possible on the systems tested.

Acknowledgements. The authors would like to thank Nina Suvanphim from Cray for her support in addressing the various issues encountered on the Cray XE6 used in this paper and installed at the Swiss National Supercomputing Centre. The work presented in this paper is supported by the "NextMuSE" project receiving funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement 225967.

References

1. MPICH2, `http://www.mcs.anl.gov/research/projects/mpich2`
2. MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE, `http://mvapich.cse.ohio-state.edu/index.shtml`
3. Alverson, R., Roweth, D., Kaplan, L.: The Gemini System Interconnect. Symposium on High-Performance Interconnects pp. 83–87 (2010)
4. Bruggencate, M., Roweth, D.: DMAPP – An API for One-sided Program Models on Baker Systems. In: Proceedings of Cray User Group (2010)
5. Gropp, W., Lusk, E., Thakur, R.: Using MPI-2: Advanced Features of the Message-Passing Interface. MIT Press, Cambridge, MA (1999)
6. Howison, M., Koziol, Q., Knaak, D., Mainzer, J., Shalf, J.: Tuning HDF5 for Lustre File Systems. In: Proceedings of Workshop on Interfaces and Abstractions for Scientific Data Storage (2010), LBNL-4803E
7. Pritchard, H., Gorodetsky, I.: A uGNI-Based MPICH2 Nemesis Network Module for Cray XE Computer Systems. In: Proceedings of Cray User Group (2011)
8. Soumagne, J., Biddiscombe, J.: Computational Steering and Parallel Online Monitoring Using RMA through the HDF5 DSM Virtual File Driver. Procedia Computer Science 4, 479–488 (2011), ICCS 2011
9. Soumagne, J., Biddiscombe, J., Clarke, J.: An HDF5 MPI Virtual File Driver for Parallel In-situ Post-processing. In: Recent Advances in the Message Passing Interface, Lecture Notes in Computer Science, vol. 6305, pp. 62–71. Springer (2010)
10. Thakur, R., Gropp, W., Lusk, E.: Optimizing noncontiguous accesses in MPI-IO. Parallel Computing 28(1), 83–105 (2002)
11. The HDF Group: Hierarchical Data Format Version 5 (2000–2011), `http://www.hdfgroup.org/HDF5`
12. UPC Consortium: UPC Language Specifications, v1.2. Tech report (2005), `http://upc.gwu.edu/docs/upc_specs_1.2.pdf`
13. Walker, D.W., Otto, S.W.: Redistribution of Block-Cyclic Data Distributions Using MPI. Concurrency - Practice and Experience 8(9), 707–728 (1996)