Performance of Parallel IO on the 5860-node HPE Cray EX System ARCHER2

David Henty
EPCC
The University of Edinburgh
Edinburgh, UK
d.henty@epcc.ed.ac.uk

***Abstract*—EPCC has recently started supporting the new UK National Supercomputer service ARCHER2, a 5860-node, 750,080-core HPE Cray EX system. In this paper we investigate the parallel IO performance that can be achieved on ARCHER2 and compare to experiences on the previous system ARCHER, a Cray XC30. The parallel IO libraries MPI-IO, HDF5 and NetCDF are benchmarked for collective writing to a single shared file, as well as the file-per-process approach for comparison. Results are obtained using a simple IO benchmark - `https://github.com/davidhenty/benchio` - which writes a large, regular, three-dimensional distributed dataset to file. We measure performance on two Lustre filesystems, one with spinning disks and the other using solid state NVMe storage. We find that although we can saturate the IO bandwidth writing multiple files, parallel performance for a single shared file is well below the expected rate. Although this appears to be because the libraries are not optimally configured for a system where a single process cannot saturate the bandwidth of one storage unit, attempts to optimise this only lead to marginal improvements.**

***Index Terms*—parallel, computing, HPC, IO, Lustre, ARCHER2**

I. Introduction

File input and output often become a severe bottleneck when parallel applications run on large numbers of processors. Simple methods such as writing a separate file-per-process or performing all IO via a single controller process are no longer feasible at scale. In order to take advantage of the full potential of modern parallel file systems such as Lustre, IO also needs to be done in parallel.

EPCC started to operate ARCHER2 [2], the UK national academic supercomputer service funded by UKRI (UK Research and Innovation), in November 2021. It is a large CPU-based HPE Cray EX system with a dedicated parallel Lustre filesystem for application IO from the compute nodes.

Our main experiences of parallel IO performance were obtained on the previous ARCHER service which was decommissioned in January 2021. ARCHER was a Cray XC30 system with a similar number of nodes to ARCHER2; however, its Lustre filesystem had very different performance characteristics comprising a much larger number of smaller, slower disks. ARCHER2 also has a substantial amount of storage not built on spinning disks but using solid state NVMe storage, which is new to the UK HPC community.

This work was financially supported by the ARCHER2 CSE grant and by the PRACE-6IP project [1] funded in part by the EU's Horizon 2020 Research and Innovation programme (2014 - 2020) under grant agreement 823767.

This paper is based on initial work done as part of the work of the ARCHER2 Computational Science and Engineering (CSE) support team, aiming to give users advice on the best parallel IO settings for their own applications. As the Lustre filesystems on the ARCHER2 HPE Cray EX have very different performance characteristics to that of the preceding XC30, previous experience may not be useful in achieving good IO rates.

II. Hardware

The hardware specifications of the two HPC systems are summarised in Table I. On both systems the physical disks - called Object Storage Targets (OSTs) in Lustre - are split across three separate filesystems. This is to provide a balance between total IO bandwidth – which is related to the number of OSTs in each filesystem – and contention for the single Meta Data Server (MDS) which controls each filesystem. The number of Lustre clients is very similar between the two systems as the number of compute nodes is practically the same (each node is a separate Lustre client), although the number of MPI processes accessing each client will be very different as ARCHER2 has more than five times more CPU-cores per node. The number of OSTs per filesystem is also very different with ARCHER2 having around four times fewer than ARCHER. The ARCHER2 OSTs are much larger in capacity, around 350 GiB each vs 25 GiB on ARCHER.

ARCHER2 also has a fourth Lustre filesystem using solid state NVMe storage. Although this is not yet open for user service, we have access to it for benchmarking purposes. The HPE data sheets indicate that the maximum achievable write bandwidth per OST is 11 GiB/s for disk and 55 GiB/s for NVMe; across each entire filesystem this gives aggregate bandwidths of 132 GiB/s and 1,100 GiB/s respectively.

III. Software

A. Benchmark

All results come from the simple *benchio* benchmark – `https://github.com/davidhenty/benchio` – which writes a large, regular, three-dimensional distributed dataset to a single shared file. It has been used for a number

	ARCHER Cray XC30	ARCHER2 HPE Cray EX
Compute		
CPU	2× 12-core Intel Ivy-Bridge	2× 64-core AMD EPYC
#nodes	4,920	5,860
#cores	118,080	750,080
network	Cray Aries	HPE Cray Slingshot
Disk		
technology	ClusterStor	ClusterStor L300
#FS	3× Lustre	3× Lustre
#OST / FS	50	12
capacity	4 PiB	13 PiB
NVMe		
technology		ClusterStor E1000F
#FS		1× Lustre
#OST / FS		20
capacity		1 PiB

TABLE I
ARCHER VS ARCHER2 HARDWARE

of previous IO benchmarking studies [3], [4], [5]. The MPI-IO part was originally developed to improve the IO performance of a cellular automaton code parallelised using Fortran Coarrays [6], hence the choice of Fortran as the language. The cellular automaton was based on large 3D integer arrays decomposed in parallel across a 3D grid of processes (or "images" in coarray terminology). IO was originally done in serial by sending all data through a single controller image which became a performance bottleneck when scaling to large system sizes. It was straightforward to call a parallel IO implementation written using MPI-IO (the coarray / MPI interface is very straightforward as local coarray data can simply be treated as a normal Fortran array), and this IO subroutine became the basis of benchio.

Parallel applications have a huge range of different IO patterns so it is impossible to cover all performance characteristics with a single benchmark. Despite its simplicity, the IO pattern of benchio is representative of any application that uses regular domain decomposition and is actually quite challenging for a parallel IO library to implement. When writing to a single shared file the domain decomposition means that the data from a single process is split into a large number of small contiguous sections of the file. As disk systems are optimised for large contiguous writes this places a heavy load on the IO library in terms of reorganising the data internally between processes prior to writing to disk; the fact that Lustre can store a single file across multiple OSTs makes this even more challenging. As an example, a 2048^3 array decomposed across 512 processes in an 8^3 grid has some 8 billion elements in total (16 million per process in a 256^3 sub-array). However, when writing to a single file, this is split into 16 million individual sections each comprising only 256 elements. Note that in benchio we always use double-precision arrays, i.e. 8 bytes per element.

Once it became clear that it would make a useful standalone benchmark, HDF5 [12] and NetCDF [13] formats were added using code previously developed for the EU-funded EUFORIA project [7]. The EUFORIA benchmark had used a one-dimensional parallel decomposition where each process owns a large contiguous section of the file, so the IO pattern was not as complicated and fast IO rates were more straightforward to achieve. The simple file-per-process approach, where each process opens its own file and no data reorganisation is required, was also added to benchio as this is useful in selectively saturating different levels of the IO hierarchy. For file-per-process, benchio does not use the MPI-IO library: it simply uses Fortran binary stream IO with `open`, `write` and `close`. We plan to add ADIOS2 [14] as an additional format but this work is still in progress (we have preliminary IO figures for ADIOS2 from other CSE work on ARCHER2).

In benchio, timing starts before the first file open and ends after the final file close. Although it would be straightforward to add parallel input benchmarks to benchio, we concentrate on output as most HPC applications write more data than they read [8]. Caching can also make the performance of parallel read more challenging to interpret.

B. Libraries and System Software

Unless otherwise stated, all performance results were produced using the programming environment PrgEnv-cray/8.0.0 which provides implementations of MPI-IO (included in the default Cray MPI), HDF5 via the module cray-hdf5-parallel (which uses MPI-IO) and NetCDF via the module cray-netcdf-hdf5parallel (which uses HDF5).

The benchmark was compiled using the Cray Fortran compiler, although this should have minimal impact on performance as all the time is spent in the IO libraries. Lustre is provided by HPE, optimised for the Shasta architecture.

IV. METHODOLOGY

Performance benchmarking any HPC application is always challenging in terms of reproducibility as components such as the interconnect are shared between multiple jobs. IO benchmarking is even more challenging as the same IO hardware is shared by all jobs using the same filesystem. As a result, it is common to obtain outlying results where IO rates are poor as other users were performing substantial amounts of IO at the same time. For this study we are aiming to understand the fundamental characteristics, performance limitations and bottlenecks of parallel IO on ARCHER2. We are therefore interested in the maximum IO bandwidth we can obtain, so we repeat measurements 10 times and take the fastest result. We take care not to run more than one IO benchmarking job at a time so we are at least never clashing with our own application. IO rates to disk typically have a standard deviation of around 10%; for NVMe the variation is less as the filesystem is not yet open for general usage.

Lustre achieves performance by having multiple OSTs in each filesystem, and allowing for a single file to be stored across multiple OSTs by partitioning it into multiple stripes. Lustre can try and ensure load-balance by using different OSTs for different files. For example, if an application uses unstriped files (each stored on a single OST) then these will automatically be distributed roughly evenly between all the available OSTs.

To investigate the effect of striping, most runs of the benchmark write to three separate directories in turn, each set to have different stripe counts using the Lustre command `lfs setstripe -c <stripe count>`:

1) *unstriped* with a stripe count of 1;
2) *striped* with a stripe count of 4;
3) *fullstriped* with a stripe count of -1.

Full striping equates to 12 stripes on the disk filesystems and 20 stripes on the NVMe filesystem.

V. File-per-process

A. IO to all OSTs

Writing a separate file from each MPI process, typically identified by its rank, is a standard approach when users first develop a parallel application. Although it is a poor approach in terms of usability when scaling to large process counts – marshalling so many separate files is awkward and reconstructing the global dataset typically requires post-processing – it does make maximum use of the full parallel bandwidth of Lustre provided there are many more processes than OSTs. The downside in terms of performance is in terms of latency: the single MDS may become overloaded with so many file open and close requests.

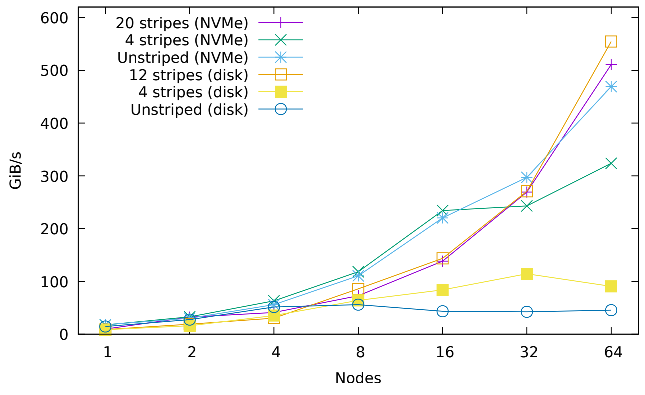

Fig. 1. File-per-process bandwidth with 128 MiB files.

The results are shown in Figure 1 where every file represents a 256^3 cube and is of size 128 MiB. Since we have 128 processes per node, each node is writing 16 GiB of data.

The aim of this test is to try and saturate all the OSTs to infer the maximum achievable bandwidth per-OST. As we are writing so many files, all the OSTs are used regardless of striping so we would expect the bandwidth be largely independent of stripe count; in fact, striping might be expected to reduce bandwidth in this case as there is more book-keeping required to work out which stripe goes to which OST.

The NVMe results largely follow this pattern with the unstriped bandwidth the highest. It appears that 64 nodes is insufficient to totally saturate the OSTs as the curve is still rising (the peak is expected to be over 1 TiB/s), but we might infer that the per-OST NVMe bandwidth is at least around 30 GiB/s (as there are 20 OSTs).

The disk results are confusing as the unstriped rates are so low despite the fact that all 12 OSTs should be available. The striped results are surprisingly high – at face value indicating a per-OST disk bandwidth of over 40 GiB/s which exceeds the quoted peak rate of 11 GiB/s. A possible explanation is that data is cached at some point in the IO software stack and we are not measuring the true IO speed to disk.

B. IO to a single OST

A more direct way of measuring the maximum OST bandwidth is to do all IO to a single OST. This can be done in Lustre by specifying a stripe count of one and selecting a specific OST with the `--stripe-index` option to `lfs setstripe`.

We observed extremely poor performance when writing from every CPU-core (128 MPI processes per node), implying that some part of the IO stack was being overloaded. We therefore benchmarked one 512^3 array per node and eight 256^3 arrays per node.

The results are shown in Figure 2 where all files were unstriped. For 8 files per node, the disk results indicate a maximum per-OST bandwidth of 12 GiB/s whereas for NVMe it is 55 GiB/s. Both these figures are remarkably close to the stated peak rates of 11 GiB/s and 55 GiB/s.

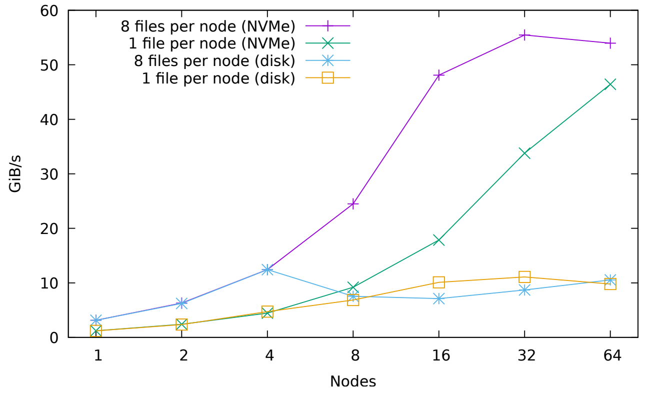

Fig. 2. File-per-process bandwidth to 1 OST with 1 GiB of data per node.

C. Bandwidth per node

One obvious feature of Figure 2 is that, for 4 nodes or fewer, the results are almost identical between disk and NVMe. This would imply that the raw file writing speed is not the limiting factor here as this speed is very different between the two filesystems. It appears that there is some other limiting factor in the bandwidth which, for small numbers of nodes, is less than the OST bandwidth.

To further investigate this we re-ran the file-per-process study of Section V-A but on a single node with a range of process counts, rather than on multiple fully-populated nodes. To compare with the previous study we used strong scaling within a node so that the total amount of data was 16 GiB per node as before. The results are shown in Figure 3.

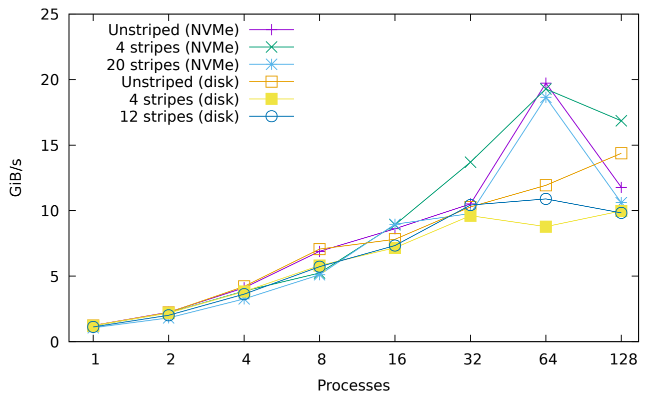

Fig. 3. File-per-process bandwidth from a single node with 16 GiB of data.

The data for up to 4 processes indicates that there is a per-process IO limit of just over 1 GiB/s. After that the data is more complicated and is not consistent with a simple per-node bandwidth limit as the disk and NVME data are different. However, it would appear that a node can sustain between 15 GiB/s and 20 GiB/s which is more than the the peak of a single disk OST but less than for NVMe.

D. *Summary*

On ARCHER, studies of file-per-process IO gave quite a simple picture [9]. Bandwidth was largely independent of striping, and aggregate bandwidths of around 15 GiB/s were achievable on all 50 OSTs, i.e. 0.3 GiB/s per OST. A single process could write at 0.5 GiB/s which is slightly more than the per-OST limit.

All the peak Lustre bandwidths on ARCHER2 are much higher as the technology is much newer. However, the balance is very different with a smaller number (12 or 20) of much faster OSTs. The picture is also not so straightforward and determining the per-OST limit is more difficult than on ARCHER, requiring us to write to a single OST rather than the whole filesystem. However, for both the disk and NVMe filesystems, it is clear that the OSTs can sustain rates in the tens of GiB/s. This is much higher than the per-process limit of around 1 GiB/s.

The major difference between the two systems is that on ARCHER2, to get anywhere near saturating the OSTs, we will need to have multiple processes writing per node; on ARCHER a single process per node was sufficient. This will become important when considering the performance of parallel libraries when writing to a single shared file, which is the main operation that is measured by benchio.

VI. MPI-IO

Having used the simple file-per-process approach to investigate the peak performance of the Lustre filesystems, we now look at parallel IO to a single shared file which is more relevant to most parallel applications. There are a number of parameters that can be varied when investigating parallel IO, such as the MPI implementation and the Lustre stripe count and size. Rather than do an exhaustive investigation for all three libraries (MPI-IO, HDF5 and NetCDF) we choose to look for the best parameters for MPI-IO then use these for the other libraries. We believe this is a sensible approach as both HDF5 and NetCDF ultimately call MPI-IO.

For all studies we do strong scaling with a global data size of $1024 \times 1024 \times 2048$ which gives a file size of 16 GiB. On a single node this gives the same amount of data per process – 128 MiB – as the strong scaling file-per-process runs (with fully populated nodes).

A. *MPI library*

We first look at MPI-IO with the default MPI settings. The results are summarised in Table II.

nodes	stripes	GiB/s
1	1	1.07
1	2	1.58
1	12	1.22
2	1	0.01
2	2	0.26
2	12	N/A

TABLE II
MPI-IO TO A SINGLE SHARED FILE ON DISK WITH DEFAULT SETTINGS

These results are very poor (for two nodes and 12 stripes the job did not complete in the time limit). We consulted HPE and they suspected that it was due to poor collective performance in the MPI library and recommended setting an environment variable that had previously been seen to improve MPI collectives for large buffer sizes: `export FI_OFI_RXM_SAR_LIMIT=64K`. this setting was used for all further studies with the default MPI.

Unlike previous Cray HPC systems, MPI can use more than one low-level transport layer. The default is OpenFabrics (OFI), but the alternative UCX library from Mellanox can also be used by swapping a couple of modules. Using UCX had previously been seen to improve collective performance for large buffer sizes.

The results are shown in Figure 4. The environment variable has significantly improved the IO, but UCX is better still. Note that the MPI-IO library hangs for 32 nodes using UCX which is why the graph stops at 16 nodes; we are currently investigating this issue. Because UCX is so much better we did not pursue attempting to optimise the `FI_OFI_RXM_SAR_LIMIT` setting.

To understand the performance, it helps to have a basic understanding of how the MPI-IO library is implemented [10]. For collective IO, a subset of MPI processes are designated as *aggregators*. These aggregators collect data from the other processes and write it to disk. By default, Cray MPI assigns one aggregator per stripe and selects aggregators on different nodes (if possible). The way the aggregators are assigned was verified by setting an environment variable – `export MPICH_MPIIO_STATS=1` – which causes the MPI-IO library to print a variety of useful statistics.

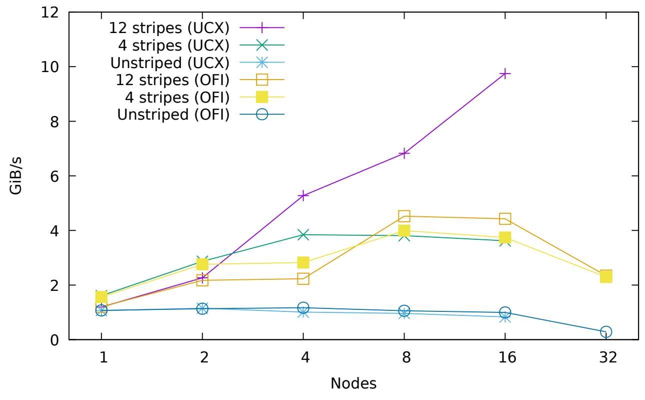

Fig. 4. Comparison of UCX and OFI for MPI-IO to disk

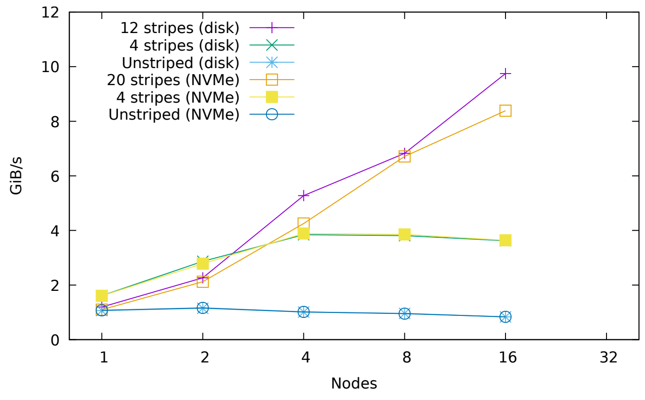

Fig. 5. MPI-IO to a single shared file using UCX

Using UCX, where the data aggregation is done efficiently and performance is limited by IO to disk rather than data transfer over the network, we therefore see performance increase roughly linearly with node count up until the maximum number of stripes. This also explains why the unstriped curve is flat as the IO is performed by a single aggregator and is capped at just over 1 GiB/s (the same figure we see for single-process Fortran write in Figure 2). For 4 stripes, IO increases up to 4 nodes and then flattens off; we assume that, for 12 stripes, the 32 node rate would be the same as 16 nodes but we are unable to obtain this data point at present.

Having only a single process performing IO to each OST is far from optimal on ARCHER2 as we have seen that the OST bandwidth is much larger than the single process limit. The reason that performance increases with stripe count at 1 and 2 nodes is because there are more aggregators than nodes and hence more than one process is writing to each OST. However, the performance does not increase linearly with the number of aggregators (e.g. only 50% faster for 4 stripes compared to 1 on a single node) indicating that there is some contention between aggregators when accessing the same OST. Scaling with node count is also not perfectly linear but it is good: for 12 stripes we would predict 8 GiB/s on 8 nodes and 12 GiB/s on 16 nodes when the actual figures are 6.8 GiB/s and 9.7 GiB/s (around 80% parallel efficiency).

We saw from Figure 2 that the NVMe OSTs were around 5 times faster, and we have 20 of them compared to 12 OSTs for the disk filesystems. However, MPI-IO performance is no better - see Figure 5 where we compare NVMe to the previous UCX results for disk. For the same number of stripes, performance is practically identical between NVMe and disk, although it is somewhat surprising that the NVMe filesystem is slower than disk at 16 nodes because we will have 20 aggregators writing rather than 12.

This is a very different situation to ARCHER where a single process was able to saturate an OST, so one aggregator per stripe was optimal. The default performance of MPI-IO on ARCHER2 is worse than on ARCHER despite the much faster OSTs.

B. *Collective IO*

In benchio, the collective MPI-IO routine `MPI_File_write_all` is used to write the data. As a collective operation, the library knows that all processes in the communicator will be calling the routine and can therefore employ a variety of optimisations to improve IO performance. Most importantly, data from different processes can be merged before being written to disk resulting in a small number of large IO transactions. For non-collective IO the data from each process has to be written individually, which for benchio results in a large number of small IO transactions. Individual processes will also have to lock the file to ensure data consistency.

It is interesting to see what improvement this makes in practice, so we did some trial runs where `MPI_File_write_all` was replaced by the non-collective routine `MPI_File_write` (the parameters to the call are all identical). The results are shown in table III.

nodes	stripes	GiB/s
Disk		
1	1	0.05
1	2	0.34
1	12	0.31
2	1	0.05
2	2	0.18
2	12	0.25
NVMe		
1	1	0.21
1	2	0.32
1	20	0.28
2	1	0.32
2	2	0.39
2	20	0.38
4	1	0.30
4	2	0.35
4	20	0.34

TABLE III
NON-COLLECTIVE MPI-IO TO A SINGLE SHARED FILE

For the disk filesystem writing to a single OST is very slow, presumably because all processes are having to issue locks to the same OST; having multiple OSTs alleviates this to some extent, although for 4 nodes the performance was so poor that we were not able to obtain results. Although locking does not seem to be so much of an issue for NVMe, Table III clearly demonstrates that collective IO is essential to scalable performance.

VII. HDF5 and NetCDF

Before trying to improve the MPI-IO performance, we look at how HDF5 and NetCDF perform compare to MPI-IO. To ensure that all IO was measured under similar conditions we benchmarked all three libraries in the same batch jobs, so the MPI-IO data will not be identical to that previously presented.

First we revisit OFI (after setting the appropriate OFI environment variable as before) – see Figures 6 and 7 which use the disk and NVMe filesystems respectively. The really surprising feature here is that HDF5 and NetCDF perform and scale better than MPI-IO, despite ultimately using MPI-IO to write to file. One possible explanation is that HDF5 and NetCDF do their own data aggregation before calling MPI-IO. Hence they are using different MPI collective routines that may be be well optimised on ARCHER2, as opposed to those collectives used internally by MPI-IO which appear to be poorly optimised (even after setting the appropriate environment variable).

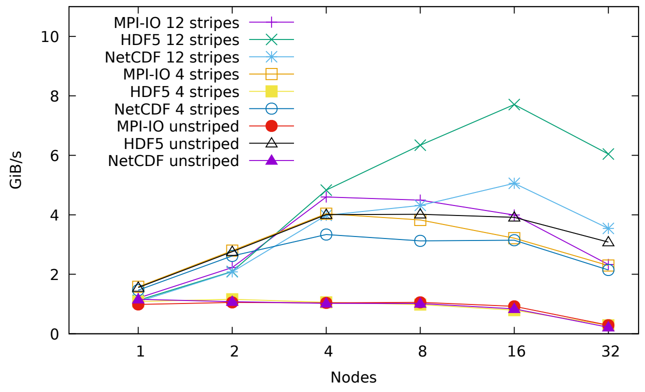

Fig. 6. Parallel IO to disk using OFI MPI

As before, the UCX results are faster and also much cleaner - see Figures 8 and 9. Here we see the expected result that, for a given stripe count, the general behaviour for all the libraries is the same but NetCDF is slightly slower than HDF5, which in turn is slightly slower than MPI-IO. This presumably reflects the additional software overheads, since NetCDF calls HDF5 which then calls MPI-IO.

A. Summary

Using the UCX version of MPI leads to reasonable scalability with increasing node count for all the libraries when writing to a single shared file. However, the absolute performance is poor, more than an order of magnitude less than the theoretical peaks. This appears to be because each OST is only being accessed by a single process on a single node, and we have

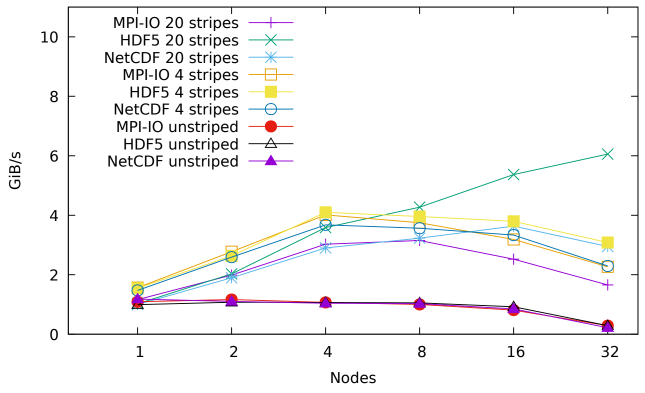

Fig. 7. Parallel IO to NVMe using OFI MPI

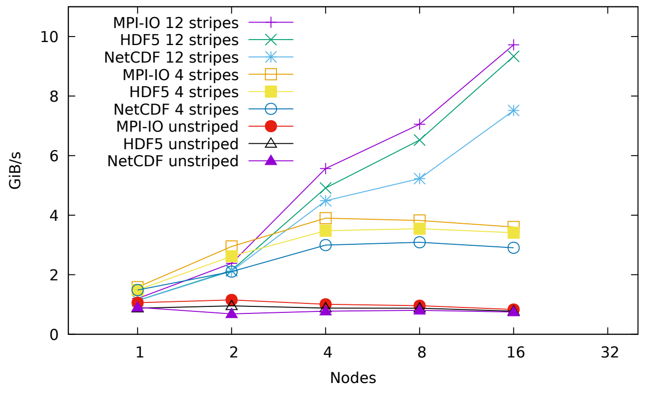

Fig. 8. Parallel IO to disk using UCX MPI

seen from the file-per-process benchmarks that that this limits performance to around 1 GiB/s per OST.

VIII. Optimising MPI-IO

We now look at whether the MPI-IO performance can be improved by tuning various library settings

A. Stripe size

We first look at varying the stripe size, which has a default of 1 MiB. When distributing a file across OSTs, Lustre divides it into many small stripes and allocates these cyclically to the OSTs. For example, for a 12 MiB file with 4 stripes (using the default 1 MiB stripe size), the first OST will store the first, fifth and ninth megabytes, the second OST the second, sixth and tenth megabytes, etc. Given the multi-gigabyte files we are working with, 1 MiB seems rather small.

We choose the largest node count possible for UCX – 16 nodes – and maximal striping, then vary the stripe size from 128 KiB to 64 MiB. The results are shown in Figure 10. It is clear that 1 MiB is actually a sensible default and the bandwidth cannot be improved significantly by varying it.

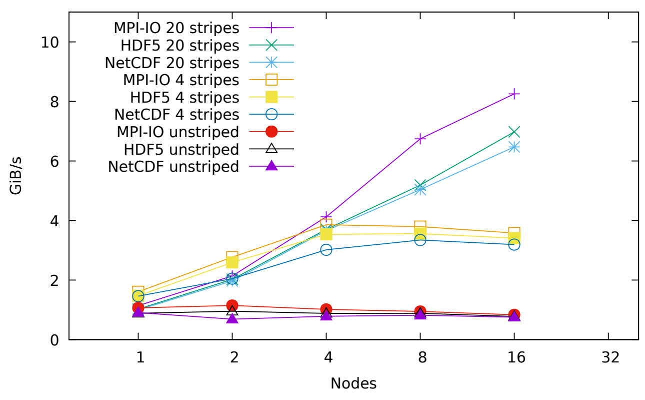

Fig. 9. Parallel IO to NVMe using UCX MPI

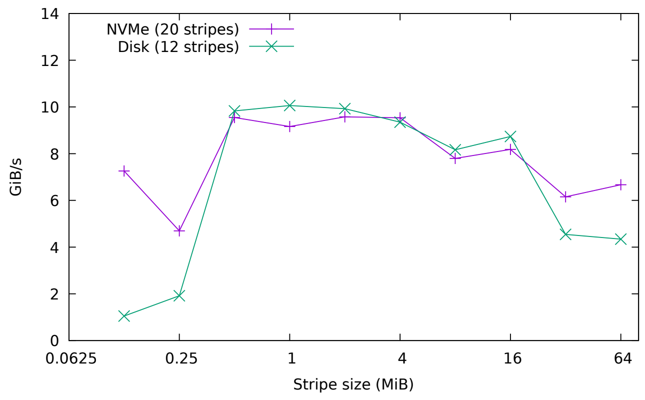

Fig. 10. Varying the stripe size for MPI-IO on 16 nodes and maximal striping.

B. Aggregator settings

The obvious improvement would appear to be having more than one aggregator per stripe. Fortunately, the number of aggregators can be controlled by the environment variable `MPICH_MPIIO_HINTS`. For example, setting this to `*:cray_cb_nodes_multiplier=2` will allocate two aggregators per stripe (double the default) for all files (specified by `*`). The fact that this has the desired effect can be checked by examining the output triggered by setting `MPICH_MPIIO_STATS=1`.

It is also possible to address the issue of contention between aggregators writing to the same OST by changing the default locking policy. By default each MPI process locks the file to prevent clashes with other processes. However, if all IO is collective, the MPI-IO library ensures that there are no clashes so this can be relaxed to a shared lock amongst all processes. This can be done by setting the hints `*:cray_cb_write_lock_mode=1` (the default locking mode is 0) and `*:romio_no_indep_rw=true`. We also look at varying the stripe size and run with 4 MiB stripes as well as the default of 1 MiB. Note that these non-default locking options *cannot be used for HDF5 or NetCDF* as both libraries perform non-collective as well as collective IO.

The results for disk output is shown in Figure 11. This is rather disappointing as none of the settings give faster IO than the default.

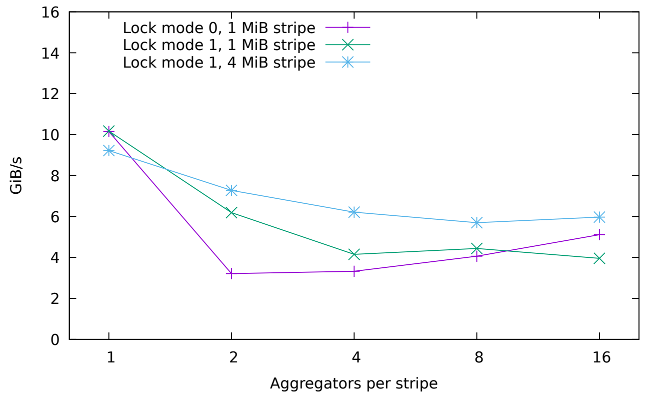

Fig. 11. Varying lock mode, aggregators and stripe size for MPI-IO to disk (maximal striping).

For NVMe the results are slightly more encouraging – see Figure 12. Although changing the number of aggregators has little effect, the relaxed locking mode improves IO by about 20%; there is also some evidence that increasing the stripe size is beneficial.

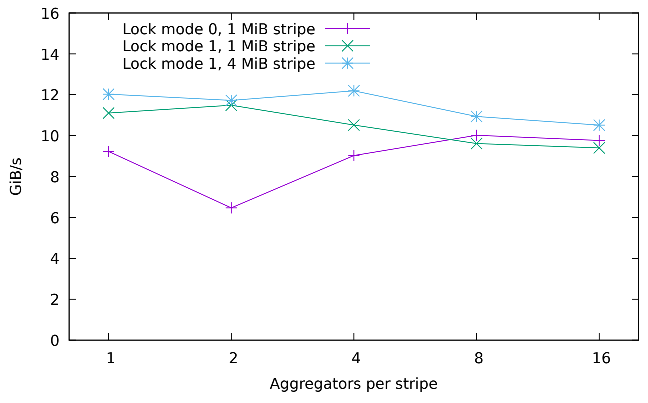

Fig. 12. Varying lock mode, aggregators and stripe size for MPI-IO to NVMe (maximal striping).

IX. Conclusions and Further Work

Overall, the performance of parallel IO to a single shared file on ARCHER2 is disappointing at present. Maximum rates are not significantly in excess of 10 GiB/s regardless of the choice of MPI-IO, HDF5 or NetCDF. Achieving even this figure requires the use of a non-default MPI library based on the UCX protocol (the default is OFI). This is roughly the same as the performance of a single disk OST, when there are 12 in each filesystem, and much less than the 55 GiB/s of a single one of the 20 NVMe OSTs. Although these are peak rates, the file-per-process benchmarks show that they are readily achievable from user code using standard IO routines. It is clear that having a single aggregator per OST, which is the default setting for MPI-IO, is far from optimal on ARCHER2 (although it was optimal on its predecessor ARCHER). However, increasing the number of aggregators has little effect on performance even when we try to minimise contention by using a shared lock. Even if the relaxed locking approach had benefitted MPI-IO, it would not be a general solution as it cannot be used for HDF5 or NetCDF.

From other work on ARCHER2 we have preliminary results for parallel IO rates from the Xcompact3D CFD application [11]. This uses a regular domain decomposition of a 3D grid similar to benchio – although constrained to a 2D "pencil" decomposition because of its reliance on FFTs – and can use MPI-IO, HDF5 and ADIOS2 for output. The MPI-IO and HDF5 results are similar to those presented here, but ADIOS2 appears to be able to achieve higher rates with its own BP4 format. Rather than always using the same file format as serial IO, BP4 sometimes produces multiple files so may be able to avoid the contention issues we see with MPI-IO. It has the same concept of aggregators as MPI-IO, selecting one per node by default, but we have not yet had time to investigate changing this. We plan to add ADIOS2 as an option to benchio so we can fully explore its potential on ARCHER2.

X. Acknowledgements

We would like to thank Harvey Richardson of HPE, and Paul Bartholomew, Andy Turner, Adrian Jackson and Shrey Bhardwaj of EPCC, for valuable discussions.

References

[1] Partnership for Advanced Computing in Europe. https://prace-ri.eu/

[2] https://www.archer2.ac.uk/

[3] D. Henty *et al.*, "Performance of Parallel IO on ARCHER", http://www.archer.ac.uk/documentation/white-papers/

[4] A. Turner *et al.*, "Parallel I/O Performance Benchmarking and Investigation on Multiple HPC Architectures", http://www.archer.ac.uk/documentation/white-papers/

[5] D. Henty, A. Jackson, C. Moulinec and V. Szeremi, "Performance of Parallel IO on Lustre and GPFS", presented at *EASC2015: Exascale Applications and Software Conference 2015*, http://www.easc2015.ed.ac.uk/program-archive/index.html

[6] Anton Shterenlikht, "Fortran coarray library for 3D cellular automata microstructure simulation", proceedings of *7th International Conference on PGAS Programming Models*, 3-4 October 2013.

[7] Adrian Jackson *et al.*, "High Performance I/O", *Proceedings of the 2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing*. 9-11 February 2011.

[8] A. Turner *et al.*, "Analysis of parallel I/O use on the UK national supercomputing service, ARCHER using Cray's LASSi and EPCC SAFE", http://www.archer.ac.uk/documentation/white-papers/

[9] A. Turner, webinar on "Parallel IO on ARCHER", 18 January 2017, slides available from https://www.archer.ac.uk/training/virtual/.

[10] R. Thakur, W. Gropp and E. Lusk, "Data sieving and collective I/O in ROMIO", *Proceedings of Frontiers '99. Seventh Symposium on the Frontiers of Massively Parallel Computation*, 26 Feb 1999.

[11] P. Bartholomew *et al.*, "Xcompact3D: An open-source framework for solving turbulence problems on a Cartesian mesh", *SoftwareX, Volume 12, July-December 2020*, https://doi.org/10.1016/j.softx.2020.100550

[12] M. Folk *at al.*, "An overview of the HDF5 technology suite and its applications", *Proceedings of the 2011 EDBT/ICDT Workshop on Array Databases*, March 25 2011, http://dx.doi.org/10.1145/1966895.1966900

[13] R. Rew and G. Davis, "NetCDF: an interface for scientific data access," in *IEEE Computer Graphics and Applications*, vol. 10, no. 4, pp. 76-82, July 1990, doi: 10.1109/38.56302.

[14] https://adios2.readthedocs.io/