Improving Nektar++ IO Performance for Cray XC Architecture

Michael Bareford, Nick Johnson, Michèle Weiland
EPCC, University of Edinburgh, Edinburgh, United Kingdom
m.bareford@epcc.ed.ac.uk

***Abstract*—Future machine architectures are likely to have higher core counts placing tougher demands on the parallel IO routinely performed by codes such as Nektar++, an open-source MPI-based spectral element code that is widely used within the UK CFD community. There is a need therefore to compare the performance of different IO techniques on today's platforms in order to determine the most promising candidates for exascale machines. We measure file access times for three IO methods, XML, HDF5 and SIONlib, over a range of core counts (up to 6144) on the ARCHER Cray XC-30. The first of these (XML) follows a file-per-process approach, whereas HDF5 and SIONlib allow one to manage a single shared file, thus minimising meta IO costs. We conclude that SIONlib is the preferred choice for single-shared file as a result of two advantages, lower decompositional overhead and a greater responsiveness to Lustre file customisations.**

***Keywords*-IO Performance; Cray XC30; Lustre; HDF5; SIONlib**

I. Introduction

Nektar++ [1] is an open-source MPI-based spectral element code widely used in the UK CFD community. The code combines the accuracy of spectral methods with the geometric flexibility of finite elements, specifically, hp-version FEM. It supports several scalable solvers for many sets of partial differential equations, from (in)compressible Navier-Stokes to the bidomain model of cardiac electrophysiology. As part of the preparations to make Nektar++ ready for Exascale platforms, we evaluate the performance benefit achieved by two parallel IO libraries. Presently, Nektar++ performs XML-based IO operations involving one file per process. This approach is expected to become untenable for core counts in excess of 10^4 cores due to the increasing burden associated with meta IO. For example, Nash et al. [2] found that the time to write a checkpoint file increases markedly for core counts greater than 1000: the communication overhead required to determine which elements are to be written by each MPI process increases by a factor of ten when the number of ARCHER nodes is increased from 32 to 256 (see Figure 6 of [2]). Another disadvantage of file-per-process IO schemes is the fact that HPC resources are often restricted by file count or inode limit, which is not an issue for codes that use a single shared file.

We measure the IO performance achieved by three IO methods, XML, HDF5 (v1.8.14) and SIONlib (v1.6.2). HDF5 [3] is a hierarchical data format that allows parallel file access. It is necessary however to record explicitly which parts of a HDF5 dataset belong to which MPI process. The SIONlib library [4] on the other hand, can record such details automatically, removing the burden of having to manage file decomposition within the application code.

Our study involves the checkpoint files produced by two test cases. One we describe as small since it produces checkpoint files that contain a moderate amount of data, approximately 2.5 MB, derived from a mesh containing 150,302 elements. The other uses a much more detailed mesh, featuring 3.5 million elements, that generates a dataset roughly 5.5 GB in size. The first test case simulates the flow of blood through an aortic arch [5] using an advection-diffusion-reaction solver to simulate mass transport, whereas the second case employs an incompressible Navier-Stokes solver to model the flow of air around a racing car. We first executed both test cases for a range of node counts, 2^n, where n is in the range $5-8$, on the ARCHER platform [6], a Cray XC-30 machine: each node has 24 cores. This enabled us to generate the various checkpoint files that could then be used by a specially written IO benchmarker.

In this paper, the problem size is the same for each node count tested: hence, we reveal the IO performance associated with strong scaling.

II. IO Methods

The performance measurement tool mentioned in the introduction is called `FieldIOBenchmarker` [2] and was written to better understand the performance costs associated with reading and writing checkpoint field files - it uses the Nektar++ `FieldIO` class, which was subclassed for each IO method covered by this paper. For example, the `FieldIOXml` class reads and writes data through the use of two routines called Import and Write. Each MPI process reads and writes to a unique checkpoint file, which stores the field definitions and field data that are handled by that process.

The `FieldIOHdf5` class supports the reading and writing of a single HDF5 checkpoint file that is accessed by all MPI processes. This file features a top-level folder called NEKTAR, which contains a sub-folder for every field type. The values of the parameters that define a field type are hashed so as to generate a unique sub-folder name — the parameter values are then added as attributes of the field type sub-folder. All other data are stored within several datasets that reside within the top-level folder. For example, every

field that an MPI process might handle consists of elements, where each element has an ID and is also associated with a set of values: hence, there are datasets for element data and element IDs. The must important dataset however concerns the decomposition; this dataset allows every MPI process to identify (via hash values) those field types it should be handling and secondly, to determine the correct offset for accessing the element datasets.

The job of creating the structure of the HDF5 file is delegated to a single *root* process that also writes the decomposition dataset having collected all the necessary metadata from the other ranks. All MPI processes then write their data to the appropriate locations within the element datasets. During the importing of HDF5 data, each MPI process uses the decomposition dataset to avoid reading parts of the element datasets that belong to other processes. The design of the `FieldIOHdf5::Import` routine incorporates a degree of flexibility that allows the user to redefine the mapping between processes and elements. This flexibility incurs a cost in the form of additional reads: specifically, the entire decomposition dataset has to be communicated to all processes.

Finally, the `FieldIOSIONlib` class does not require decompositional data to be recorded explicitly. The SIONlib library itself records which MPI processes have written which data to the checkpoint file: this permits each process to simply loop over the fields it handles, writing out or reading in the field definition, element IDs and element data in turn. It is also possible for an MPI process to impersonate another rank when opening a SIONlib file, permitting element redistribution.

The size of the checkpoint file will vary with core count and IO method. For example, SIONlib checkpoint files are significantly larger than their HDF5 counterparts. This is because the file format can be more tightly controlled using the HDF5 method and so the resulting file is *leaner*: from 768 to 6144 cores, the size of the small HDF5 checkpoint files increases slightly from `24.7` to `25 MB`. For the large test case, the different HDF5 file sizes fluctuate closely around `5.48 GB`. The SIONlib checkpoint file is formatted according to the number of MPI processes that write to it. The space reserved for an MPI rank is divided into chunks, where each chunk is a multiple of the file system block size. ARCHER uses a Lustre file system with a block size of 65,536 bytes. The SIONlib chunks are always one block in size for the small (aorta) test case, whereas the larger (racing car) checkpoint files require chunk sizes of 110, 55, 29 and 16 blocks for node (core) counts of 32 (768), 64 (1536), 128 (3072) and 256 (6144) respectively. This arrangement guarantees that the section of the checkpoint file assigned to any MPI process will not cross a file system block boundary. The downside however is that SIONlib files will contain some padding. This is particularly noticeable with the small checkpoint file: the size is approximately `50.4 MB` for 32 nodes and then doubles for each successive doubling of node count. The size of the large SIONlib checkpoint files does not rise with node count, but, these files are twice as big as their HD5 counterparts, coming in at around `11 GB`.

III. Results

Sets of scaling runs were performed for each IO method, where each run recorded the time taken to read and write the data. Both IO operations were performed ten times for each core count, allowing an average value for execution time to be plotted. The `FieldIOBenchmarker` tool takes a parameter that specifies how many tests are to be performed. Setting this parameter to ten can lead to caching effects that result in fast file access times; to counter this unrealistic circumstance, the count parameter was set to one and `FieldIOBenchmarker` was instead called ten times from within the submission script. Further, the submission script was designed such that each iteration exercised all three IO methods. Having each IO method tested from *separate* scripts running at different times could make comparisons difficult since the overall file system load is expected to be varying continuously. All of this performance data, containing 240 time measurements (4 core counts $\times$ 3 IO methods $\times$ 2 IO operations $\times$ 10 tests), was then bundled into a single dataset. Similarly formatted datasets were then compiled on subsequent days in order to sample how Nektar++ IO performance could be affected by the different file system loads routinely experienced by ARCHER.

Please note, the read and write times presented here are taken from the perspective of Nektar++, and therefore do not just cover the low-level IO operations, but also the necessary housekeeping required to initialise the data structures for all MPI ranks.

The Lustre file system on ARCHER controls access to a set of 48 object storage targets (OSTs) and has a theoretical peak performance of `30 GB/s` [7]. By default, each file is split into 1 MiB stripes and then stored on one 1 OST. This is fine for codes using one file per process; whereas a single shared checkpoint file should in general have its stripe count set to -1, allowing use of all available OSTs. However, we found that when file sizes are sufficiently small the best IO performance is achieved with the default Lustre settings.

HDF5 nominates a set of MPI ranks to act as data aggregators: those processes collect the data from all the other ranks before writing to file. The number of HDF5 file writers was 48 for the core counts tested. SIONlib also funnels data to designated writers (or collectors) assuming one has opened the file in collective mode; although, we found that the number of SIONlib writers was fixed at the lower value of 32. Furthermore, using a *collective* SIONlib configuration would not be sufficient to yield the best performance, since each writer actually performs many writes instead of just one. As with HDF5, a SIONlib writer

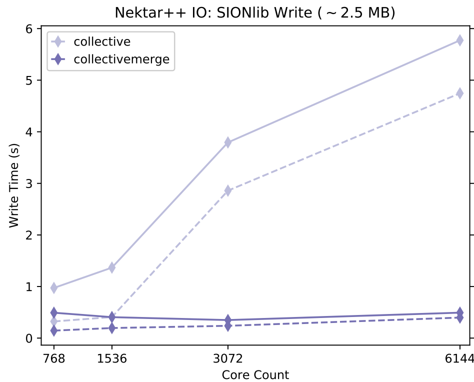

Figure 1. Scaling plots of average time (solid line) spent in `FieldIOSIONlib::Write` for two SIONlib file access modes, `collective` (pale purple) and `collectivemerge` (purple). The dashed lines indicate the minimum times achieved.

gathers data from other MPI ranks, but then it writes each rank's data to the part of the checkpoint file reserved for that rank. Thus, for high core counts, each SIONlib writer will perform thousands of writes across different locations. The solution is to add the `collectivemerge` tag to the mode string that is used to open the SIONlib checkpoint file. This forces each writer to output the collected data to one part of the file only; the part reserved for that writer's rank. Figure 1 shows the dramatic improvement gained by using collective merged writes.

Figure 2 presents the results generated by the small checkpoint files. It shows that SIONlib achieved the fastest read/write times for all core counts, but, although the majority of the data points are close to the averages (enlarged data points) there are many outlying (i.e., slower) readings also. For example, the SIONlib results recorded for a core count of 6144 contain two *extreme* outliers that are pointed to by the vertical chevrons in Figure 2. A data point is judged to be an outlier if it is beyond a certain distance from the top of the third quartile. This distance is defined as twenty times the inter-quartile range (IQR) for the read data and 12 IQR for the write data. The two SIONlib write outliers are so extreme that the mean SIONlib write time for 6144 cores is in fact greater than the XML and HDF5 averages; this is despite the fact that the fastest writes were recorded by SIONlib. If we adjust the average read/write times by discounting these extreme values, we see that SIONlib is the best performing IO method, as shown in the bottom plots of Figure 2.

The effects of file system contention are clearly seen in the scatter plots presented in this paper. It is necessary therefore to collect several datasets at different times within a 7-day period; such a procedure should guarantee a representative sample of the workloads experienced by the ARCHER system. Contrary to Nash et al. [2], we find that the deterioration in XML write speeds is closer to a factor of two rather than ten: a simple explanation here is that the earlier result had been influenced by outliers.

We repeated the measurements discussed above but this time we used a more industrial test case, one that produces checkpoint files approximately two thousand times larger than those generated by the aorta mesh. For the large test case, the SIONlib library continues to lead when it comes to reading data, but it is now the slowest of the three IO methods for writing data, see Figure 3 — although, SIONlib manages to beat HDF5 at 6144 cores it is nearly three times slower than the XML average. In fact, XML is now the fastest writer for all core counts (Figure 3, bottom).

We next investigated the impact of changing the Lustre settings for the checkpoint files. Just to recap, the default settings (stripe count/size = 1/1MiB) were used for the small checkpoint files regardless of IO method. We then changed the Lustre stripe count to -1 for the single-shared file IO methods (HDF5 and SIONlib) for the large test case, enabling a checkpoint file to be striped across all available OSTs, and compared the IO performance (Figure 3). As noted previously, the actual writing and reading of data, is handled by a (relatively) small group of processes: 48 in the case of HDF5 and 32 for SIONlib — all other processes simply send their data to the pool of designated writers. It seemed sensible then to enlarge the Lustre stripe size for the large checkpoint files to a value roughly equivalent to the amount of data handled by each writer, which turns out to be `256 MiB` for SIONlib and `128 MiB` for HDF5. In effect, each HDF5/SIONlib writer could now write all of their data within one stripe, allowing each writer to be assigned to one unique OST.

The result of these adjustments is that the HDF5 read times improved such that XML is now the slowest reader on average (Figure 4). The HDF5 *write* performance however is more or less unchanged. On the other hand, the SIONlib write times have decreased noticeable, producing times comparable to XML.

IV. Summary and Conclusions

The `FieldIOBenchmarker` tool has provided sets of results that show the read and write times for Nektar++ checkpoint files using three IO methods, XML, HDF5 and SIONlib. The first of these methods (XML) maintains one checkpoint file per MPI process, whereas the other two use a single shared checkpoint file.

We find that the relative performance of the IO methods does depend on the size of the checkpoint file. The results for the small `2.5 MB` checkpoint file exhibited significant scatter: it was necessary to annotate the corresponding plots to indicate the presence of extreme outliers (Figure 2). These extreme values (8 out of 720) were recorded for all IO methods; hence, we felt justified in recalculating the mean performance with the outliers removed. SIONlib achieved

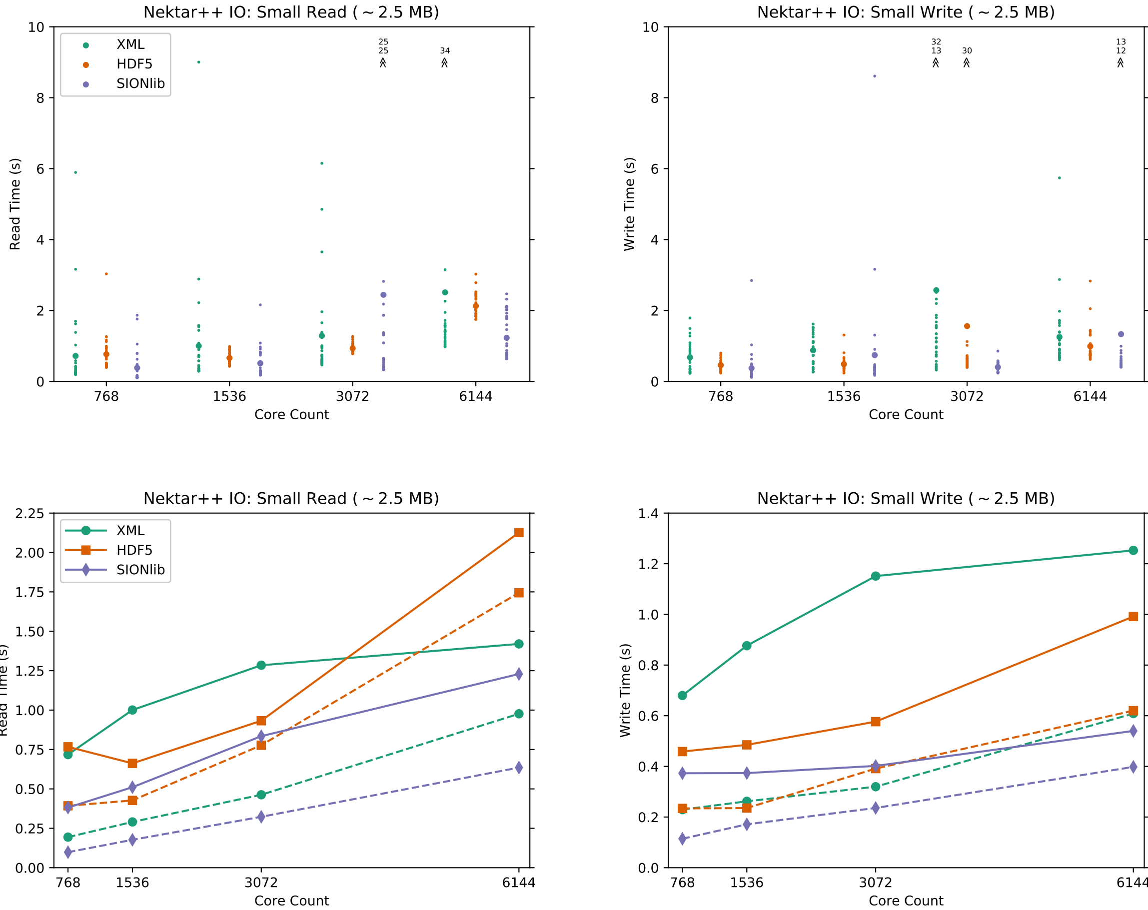

Figure 2. Read and write times for the small (2.5 MB) Nektar++ checkpoint files. Along the top row are scatter plots of read (left) and write (right) times from three datasets (720 timing measurements in total) compiled on separate days. The data covers four core counts for three IO methods, XML (green), HDF5 (red) and SIONlib (purple). The large data points represent average values. Some of the datasets contained extreme outliers, which are indicated by vertical chevron pairs with the actual outlier values printed immediately above, see text for how these outliers were identified. The mean (solid) and minimum (dashed) read and write times for the aforementioned datasets are shown in the bottom row. Note, the mean calculations in the bottom plots exclude any outliers displayed in the top row.

the fastest IO times for all core counts. XML was the slowest at writing (2.3 times the SIONlib result), but HDF5 showed the poorest read performance (1.7 times slower than SIONlib).

We should also mention that setting the stripe count to -1 for the single-shared file IO methods worsened the IO speeds for HDF5 and SIONlib, indicating, perhaps, that we had reached the core count limit for the partitioning of the aortic arch mesh [5]. This motivated further IO benchmarking but this time using *5.5 GB* checkpoint files, which were sufficiently large to justify using a stripe count of -1, permitting the checkpoint data to be striped across all available OSTs (45 on the ARCHER system) — the stripe size remained at 1 MiB. Another benefit of handling greater data volumes was the disappearance of extreme outliers (Figures 3 and 4), which had complicated the previous results.

Again, SIONlib was the clear winner for reading, with the average read taking around a quarter of the HDF5 and XML times. The picture was very different for file writing however (see Figure 3): XML was clearly the fastest with a mean write time of 1.6 seconds at 6144 cores, followed by SIONlib at 6.8 s and HDF5 at 8.5 s. In response, we increased the stripe size for SIONlib and HDF5 to 256 MiB and 128 MiB. The intention here was to create the possibility for each SIONlib/HDF5 aggregator to read/write their data to a dedicated OST using a single stripe. These changes improved the HDF5 read speeds by over 30%. The HDF5 write performance was more or less unchanged; conversely, the mean SIONlib write time (at 6144 cores) had dropped by two thirds to 2.4 seconds (Figure 4). The *minimum* write times at 6144 cores for SIONlib and XML were even closer

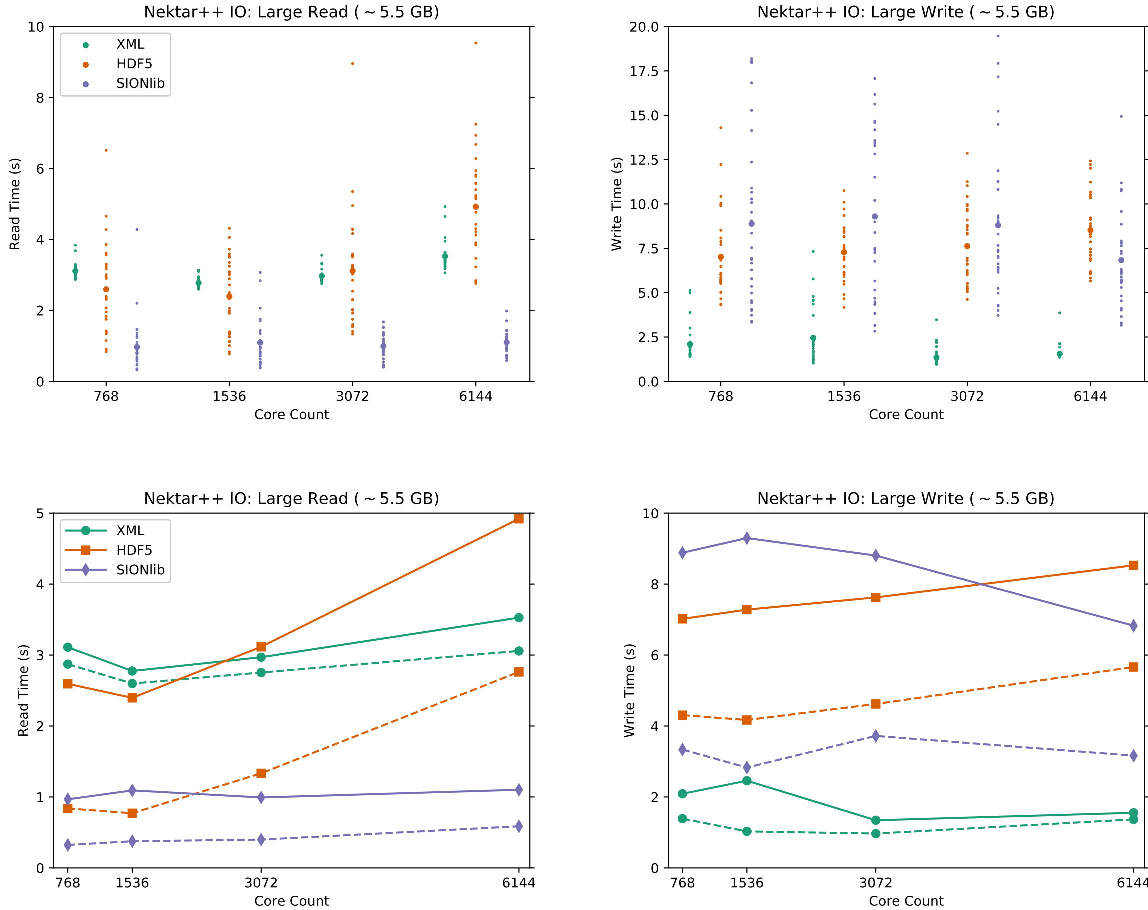

Figure 3. Read and write times for the large (5.5 GB) Nektar++ checkpoint files. Along the top row are scatter plots of read (left) and write (right) times from three datasets (720 timing measurements) compiled over several days. The data includes timings taken at four core counts for three IO methods, XML (green), HDF5 (red) and SIONlib (purple). The large data points represent average values. The mean (solid) and minimum (dashed) read and write times for the aforementioned datasets are shown in the bottom row.

than the mean results, being 1.5 and 1.3 seconds respectively.

When opening HDF5 files it is possible to specify additional access properties; unfortunately, none of the properties described in the HDF5 User's Guide ([3]) related to Lustre stripe sizes, so there seems to be no obvious way for allowing HDF5 to take advantage of the stripe size customisation. It might be possible however to improve the HDF5 read performance. We noted in Section II that the design of the HDF5 read routine required all processes to read the entire decomposition dataset. In reality, HDF5 ensures that only a subset of the MPI ranks (the aggregators) actually read the dataset, which is then communicated to the other processes. Nevertheless, the format of the HDF5 file could be altered such that every MPI rank could directly access its own dataset offsets, but one must also bare in mind that a checkpoint read is generally performed just once during a simulation making this code improvement a low priority task.

We have demonstrated that SIONlib is the superior single-shared file IO method compared to HDF5 within the context of Nektar++ checkpoint file operations. SIONlib performance is expected to exceed XML for higher ($> 10,000$) core counts due to the increasing burden of meta IO operations. This prediction is foreshadowed by the write times associated with the aorta checkpoint file (Figure 2, right): at 6144 cores, the data handled by each process is no more than half a KB and so, a large part of the time spent writing XML files will be due to meta IO, making XML slower than SIONlib.

Maximising the SIONlib write performance requires the use of the `collectivemerge` mode when opening the checkpoint file. This means that it is now less convenient to re-read the checkpoint data: reading in *merged* data requires that the original writers self identify and then partition the data stored in their area according to the original source, i.e., MPI rank. Therefore, should SIONlib

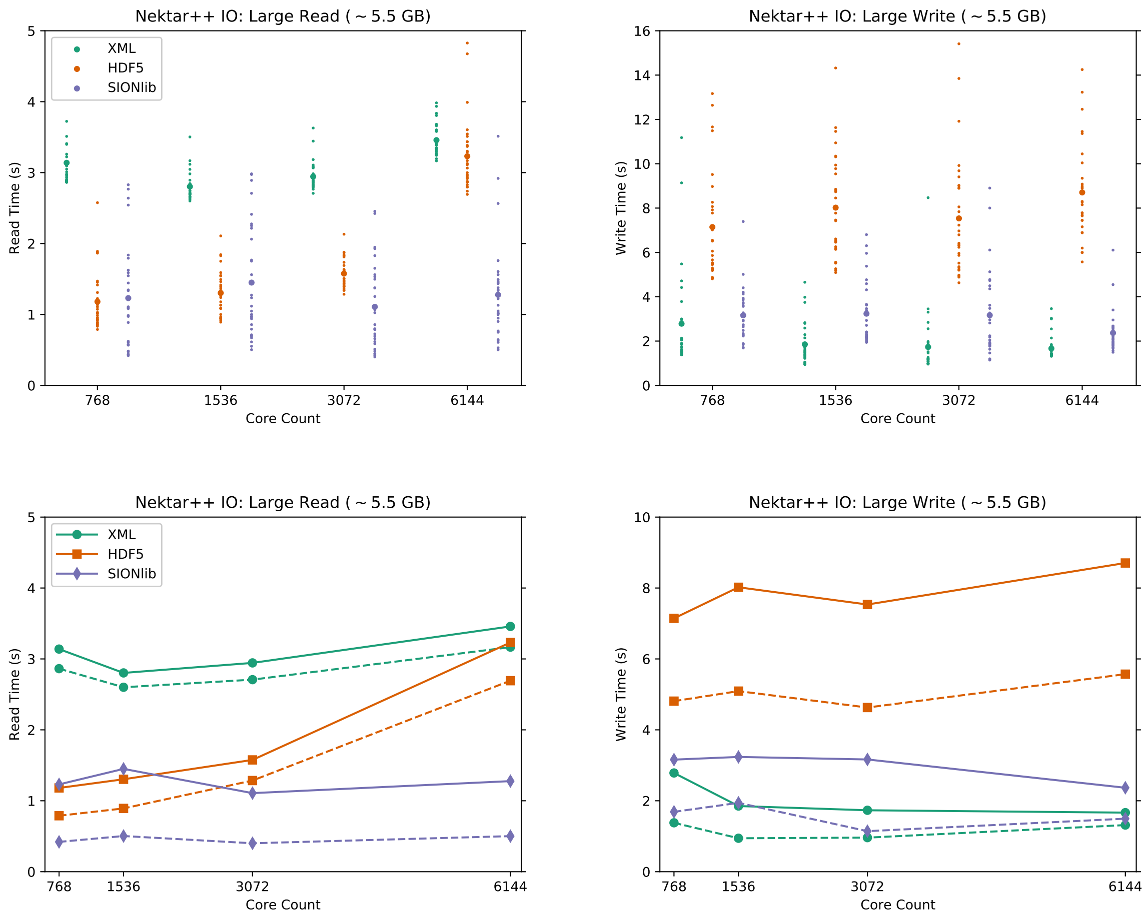

Figure 4. This figure follows the format of Figure 3, but this time the Lustre custom stripe sizes have been customised for the HDF5 (128 MiB) and SIONlib (256 MiB) checkpoint files.

be preferred over HDF5, it will be necessary to extend the `FieldIOSIONlib` class to allow reformatting of a merged checkpoint file in preparation for a simulation restart. Fortunately, any such reformatting would be a one-off cost.

Lastly, the reader should note that the IO performance of the `FieldIOHdf5::write` routine is hampered somewhat by the need to collate decompositional data. If one assumes a static element partition (i.e., no load balancing during the simulation) then the content of the decomposition dataset would only need to be written once; this improvement should bring the HDF5 performance in line with SIONlib.

REFERENCES

[1] C. Cantwell, D. Moxey, A. Comerford, A. Bolis, and G. R. et al., "Nektar++: An open-source spectral/hp element framework," *Computer Physics Communications*, vol. 192, pp. 205–219, July 2015.

[2] R. Nash, S. Clifford, C. Cantwell, D. Moxey, and S. Sherwin, "Communication and i/0 masking for increasing the performance of nektar++," Edinburgh Parallel Computing Centre, Embedded CSE eCSE02-13, May 2016. [Online]. Available: https://www.archer.ac.uk/community/eCSE/eCSE02-13/eCSE02-13-TechnicalReport.pdf

[3] HDF5, *HDF5 User's Guide*, release 1.10.0 ed. University of Illinois, USA: The HDF Group, March 2016. [Online]. Available: https://www.hdfgroup.org/HDF5/doc/UG/HDF5_Users_Guide-Responsive%20HTML5/index.html#t=HDF5_Users_Guide%2FHDF5_UG_Title%2FHDF5_UG_Title.htm

[4] Julich Supercomputing Centre (JSC). (2016) Sionlib v1.6.2 - scalable i/o library for parallel access to task-local files. Web site. [Online]. Available: https://apps.fz-juelich.de/jsc/sionlib/docu/index.html

[5] P. E. Vincent, A. M. Plata, A. A. E. Hunt, P. D. Weinberg, and S. J. Sherwin, "Blood flow in the rabbit aortic arch and descending thoracic aorta," *Journal of The Royal Society Interface*, vol. 8, pp. 1708–1719, May 2011. [Online]. Available: http://rsif.royalsocietypublishing.org/content/8/65/1708

[6] EPCC. (2015) Archer user guide. Web site. [Online]. Available: http://www.archer.ac.uk/documentation/user-guide/

[7] ——. (2017) Parallel i/o performance benchmarking and investigation on multiple hpc architectures. Web site. [Online]. Available: http://www.archer.ac.uk/documentation/white-papers/parallelIO-benchmarking/ARCHER-Parallel-IO-1.4.pdf