eCSE-0302 Final Report

Paul Bartholomew (EPCC), Charles Moulinec (STFC),
Michèle Weiland (EPCC), Sylvain Laizet (Imperial College London)

April 2023

Abstract

This report presents the work conducted for the ARCHER2 eCSE-0302 project. The project goal is to address the I/O bottleneck in the Xcompact3D CFD application by facilitating user defined *in-situ* analyses. This was achieved by first adding the ADIOS2 framework as an optional backend to the 2DECOMP&FFT library which Xcompact3D is built upon, providing run-time configurable I/O. The I/O configurations provided by ADIOS2 include redirecting the I/O to a concurrently running program instead of to disk. By also adding ADIOS2 to the Py4Incompact3D postprocessing tool, the data can be processed before writing to disk, limiting the volume of data written *vs* writing the entire 3-D solution field for later postprocessing which would additionally incur I/O costs to load the data. To be a practical solution project included work to parallelise the Py4Incompact3D tool. Testing disk-based I/O using the ADIOS2 backend showed it was capable of better scaling than was achieved by the current MPI-IO solution within 2DECOMP&FFT, which will be of benefit to other users of the 2DECOMP&FFT library. The Py4Incompact3D tool was successfully parallelised by wrapping the 2DECOMP&FFT library, demonstrating excellent scaling, allowing it to be used in larger cases than was previously practical. With the addition of ADIOS2 Py4Incompact3D was successfully coupled with a running Xcompact3D simulation on ARCHER2, performing the analysis *in-situ*. Scaling studies of the coupled Py4Incompact3D/Xcompact3D analysis showed performance comparable to Xcompact3D running in isolation until a 4:1 ratio of compute resources assigned to Xcompact3D over Py4Incompact3D was reached when load imbalance causes the coupled setup to become bottlenecked. This load balance crossover point was observed for a range of problem sizes and process counts up to a 513^3 mesh size running on 4,096 Xcompact3D MPI ranks (for 1,024 Py4Incompact3D MPI ranks), beyond which the effects of load imbalance were observed.

Keywords: *Parallel I/O; CFD; in-situ analysis.*

1 Introduction

This project seeks to address the I/O bottleneck for Xcompact3D [1, 2] with potential further benefit for the broader community building on the underlying 2DECOMP&FFT library [3]. It is well known that I/O can represent a significant bottleneck in the HPC workflow, yet much work has focused on the compute aspect resulting in hardware and software that is heavily imbalanced in favour of compute performance. To illustrate the issue, Xcompact3D was benchmarked on the ARCHER2 early-access system and Fulhame Arm cluster at EPCC using a 1025^3 mesh-node problem size either as a purely computational problem or additionally performing I/O per timestep with a total of 32.4 GiB of data written per timestep, on Fulhame the SSD filesystem was used with maximum Lustre striping, whereas the ARCHER2 early-access system being new was left with default settings. As shown in *fig.* 1 the performance of pure compute runs follows near-ideal scaling on both machines, whereas performing I/O either rapidly curtails the scaling behaviour or

even anti-scales[1].

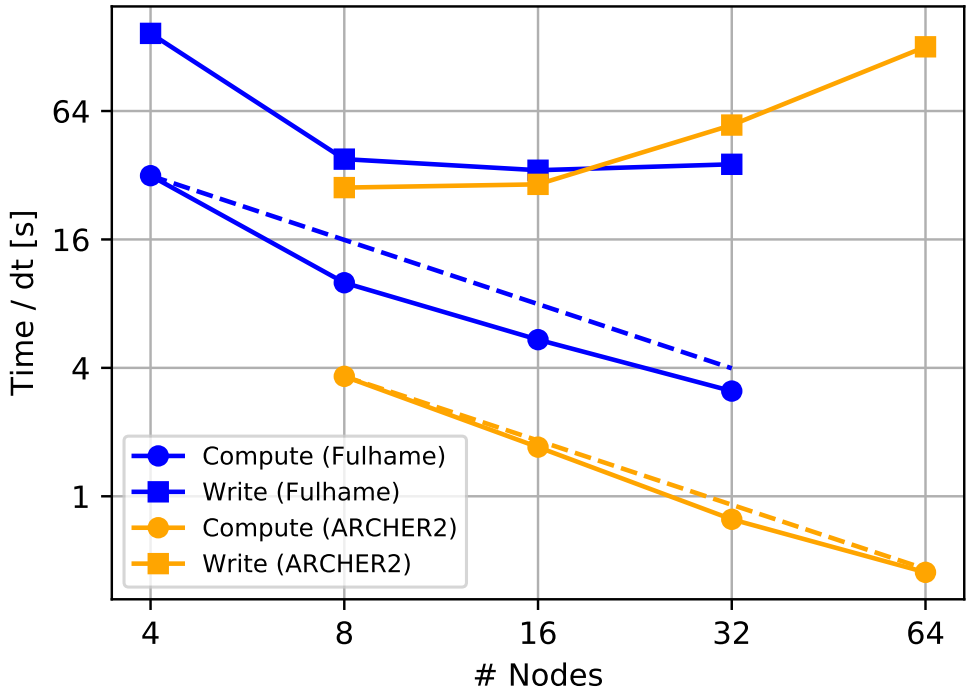

Figure 1: Timings of Xcompact3D with and without I/O per timestep. Note that the ARCHER2 system has 128 cores per node whereas Fulhame has 64 cores per node, both in a dual-socket configuration.

Ultimately computational science aims to answer some question through simulation, therefore data must be somehow analysed requiring either I/O for later analysis or performing analysis *in-situ*, reducing I/O to only the values of interest. In this project ADIOS2 [4] is integrated into the Xcompact3D/2DECOMP&FFT I/O system, giving access to a range of parallel output formats (HDF5 [5], BP4/5 [4]) and enabling runtime I/O configuration. Prior to this project a post-processing toolbox Py4Incompact3D [6] was developed to bring together multiple ad-hoc analysis scripts written in Python3, and facilitate analysing data produced by Xcompact3D and ensure consistency between the numerical methods used in the simulation and later analysis of data. In addition to providing access to various file formats, ADIOS2 can redirect I/O to another process running in parallel to the primary simulation, this will enable Py4Incompact3D to be run in parallel to Xcompact3D allowing *in-situ* analysis code to be moved out of the main Xcompact3D code, simplifying implementation of new analyses by users and improving the software engineering of the Xcompact3D project by removing hard-to-distribute problem-specific code.

2 ADIOS2 in Xcompact3D

The first work package of the project was to add an ADIOS2 I/O backend to Xcompact3d/2DECOMP&FFT. A primary goal of the project was to support existing use cases of Xcompact3d, allowing users to continue using the original MPI-IO backend, and maintain the existing I/O interface as far as possible. To do so, ADIOS2 calls were added to the existing I/O subroutine bodies and selected at compile-time via preprocessor macros as shown in listing 1.

[1]We would like to emphasise that the ARCHER2 results shown in this figure were obtained on the early-access system and so may not represent the full production system.

Listing 1: Code illustrating compile time selection of I/O backend.

```
#ifndef ADIOS2
  !! Existing MPI–IO code ...
#else
  !! New ADIOS2 code
#endif
```

There were, however a few changes necessary to support both I/O backends:

1. "Pre-registration" of I/O dataset(s): ADIOS2 requires informing the I/O engine ahead of time about the dataset(s) to be written (sizes, data type, *etc.*) which is not required by the MPI-IO backend. To support both, the new pre-registration subroutine in 2DECOMP&FFT is simply a `noop` in the MPI-IO backend, and an error message is printed by the ADIOS2 backend if attempts are made to write a variable without pre-registering.

2. The current MPI-IO backend supports writing single variable files (for solution variables) and multiple variable files (for checkpointing). One potential performance benefit in ADIOS2 is to use buffered/deferred I/O *i.e.* all I/O should look like multiple variable files. Supporting the existing MPI-IO behaviour and also achieving the maximum performance possible from ADIOS2 meant rewriting all I/O to assume it was writing multiple variables to a file - if it was already `open`. This allowed a simplification of the 2DECOMP&FFT I/O interface, whereas previously the user code was responsible for maintaining an open file handle, now the 2DECOMP&FFT library would maintain a registry of open file handles (or engines for ADIOS2) allowing single variable file MPI-IO by immediately closing the file upon read/write.

3. The existing 2DECOMP&FFT I/O interfaces perform a copy to a temporary variable to (for example) support compression via lowering the precision of output data. Performing I/O with temporary variables however breaks the ADIOS2 deferred I/O memory model with the deferred data being potentially overwritten before the I/O occurs. The 2DECOMP&FFT I/O interface was therefore rewritten to use the original array when the simulation and I/O precision are the same and the choice made not to support reduced-precision I/O when using the ADIOS2 backend, with ADIOS2 supporting compressed I/O through various compression libraries as an alternative.

    - Similarly, when variable a is copied between grids to perform I/O on the main grid it is interpolated into a temporary array defined on the main grid, in this case it becomes necessary to force a `flush` operation in the ADIOS2 backend to prevent data corruption. This situation occurs in Xcompact3D when writing the pressure field which is defined at cell centres, for visualisation purposes it is interpolated to the main grid nodes before writing.

Our main goal with adding ADIOS2 is to gain access to the *in-situ* possibilities it offers, however we also want to maintain a reasonable fraction of the performance attained by the existing MPI-IO backend. To test this the Taylor Green Vortex (TGV) example case was used as a benchmark, performing I/O every timestep for a range of problem sizes. This benchmark was run on ARCHER2 using MPI-IO and ADIOS2 with the average time per timestep used to evaluate the performance. In the first instance, smaller problem sizes (513^3 mesh nodes, ≈ 1 GiB per field) were investigated. For each backend considered the degree of parallelism presented to the application by the I/O system (referred to as "*stripes*" in Lustre nomenclature) was varied from the minimum to maximum values available in the ARCHER2 system with the timings achieved shown in *fig.* 2. From this it can be seen that MPI-IO performance is strongly dependent on the number of stripes, whereas ADIOS2's performance using the ADIOS2-native BP4 format is near-independent of the number of stripes, eventually outperforming MPI-IO with strong scaling of the problem. This performance trend is due to ADIOS2/BP4 using a multiple, rather than single, file output format meaning that there is potential for greater I/O parallelism without the need to coordinate locking files for parallel access. Also contributing to the performance is the ADIOS2 library's default behaviour to perform write agglomeration per node (as seen by performance scaling with nodes) whereas the MPI-IO implementation provided by Cray's MPI library

on ARCHER2 defaults to write agglomeration per Lustre stripe (maximum 12 stripes on ARCHER2). The HDF5 performance attained via ADIOS2 in this case however is disappointing, and it is not entirely clear why, as HDF5 is itself built atop the MPI-IO library and is therefore expected to trend towards the same performance - there is the additional layer of indirection introduced by ADIOS2, and it is possibly with some further configuration this could be improved, however the impressive out of the box performance of BP4 showed ADIOS2 could keep pace with or exceed MPI-IO.

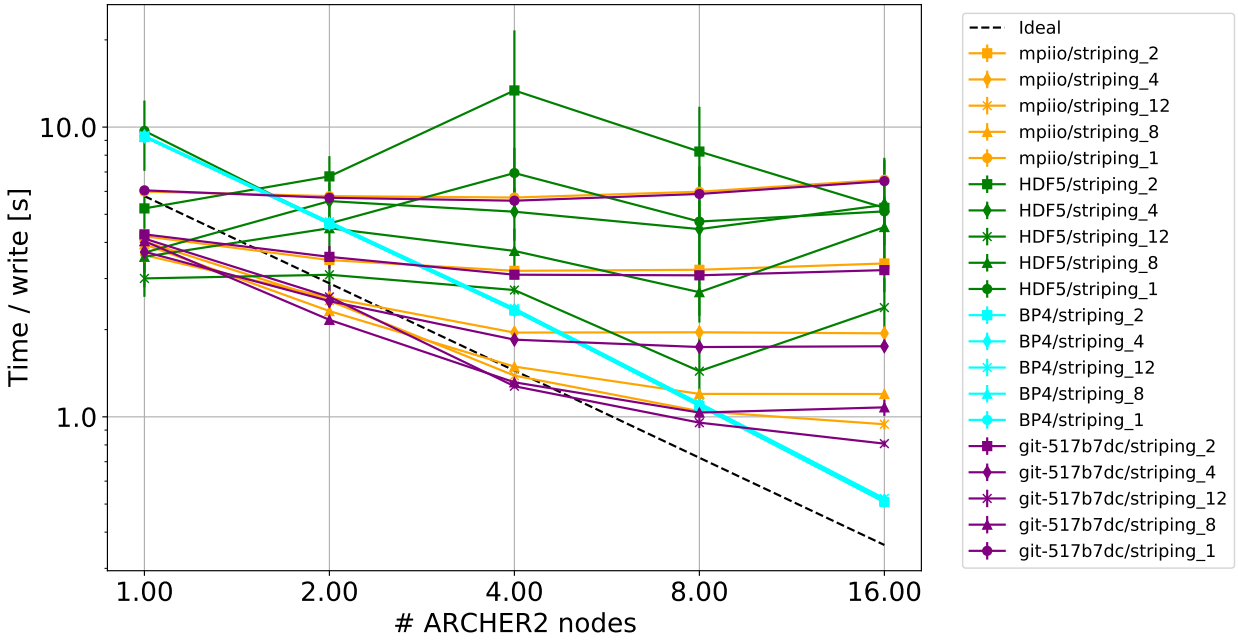

Figure 2: Comparing performance of different I/O backends in Xcompact3D with different Lustre stripe counts. The data set labelled `git-517b7de` is nominally the same as `mpiio` but is from a commit prior to adding the ADIOS2 backend to check the original `mpiio` performance has not been affected. The black dashed line shows the ideal scaling trend, data running parallel to this has ideal scaling behaviour.

A larger case (1025^3 mesh nodes, $\approx$ 8 GiB per field) based on the same example was also tested at greater core counts, using an updated version of ADIOS2. Based on the prior tests showing the dependence on stripe count for MPI-IO, and BP4's independence of this parameter, these tests were performed at the maximum stripe count only (HDF5 was not considered). *Figure* 3 shows the minimum bandwidth achieved across timesteps within a run. As can be seen the bandwidth achieved by the ADIOS2 formats is consistently higher than that of MPI-IO from 64 to 1024 ARCHER2 nodes (8,192 to 131,072 cores), based on the network bandwidth to the disk servers a maximum of approximately 144 GiB/s would be the expected maximum, and the BP5 format is able to maintain performance approaching this value, whereas BP4 shows generally lower performance, and the MPI-IO performance achieved is significantly below this value. This test, performed later in the project, also demonstrated an additional benefit of adopting a library approach to I/O such as ADIOS2: by updating the library the new BP5 format became available without any code changes and with it improved I/O performance.

3 Parallelising Py4Incompact3D

The Py4Incompact3D postprocessing tool box was initially written with workstation use in mind and was entirely serial. This already presented a problem when analysing large simulation datasets due to either memory or time required and would make running Py4Incompact3D alongside Xcompact3D for an *in-situ*

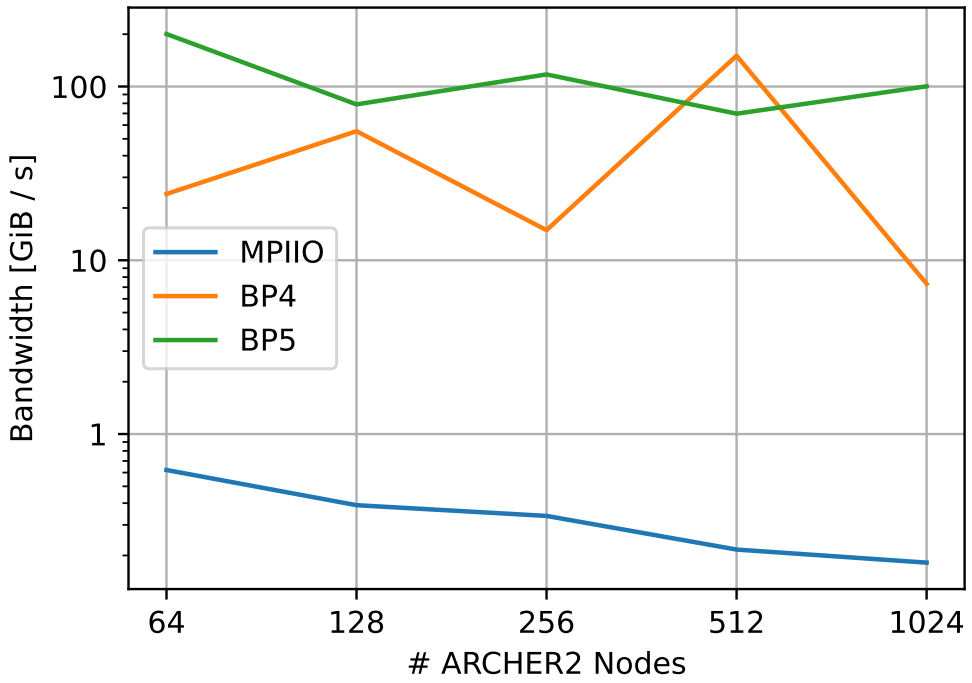

Figure 3: Measuring bandwidth of I/O achieved by Xcompact3D on up to 1024 ARCHER2 nodes (131,072 cores).

analysis impractical at scale. As Py4Incompact3D is based on the numerical methods used in Xcompact3D it was expected that the same approach to parallelisation in Xcompact3D (the 2-D pencil decomposition) should also be effective in parallelising Py4Incompact3D. To implement this a Python wrapper was written for the 2DECOMP&FFT library distributed with Xcompact3D, exposing the initialisation subroutine, methods to query the grid dimensions in a given pencil, and the various `transpose_X_to_Y()` subroutines for performing parallel data transposes to orient pencils along different coordinate axes. The wrapper is written in Fortran90 and uses `f2py` to compile and build the wrapper, it was decided that to make interfacing with Python codes (*i.e.* Py4Incompact3D) as simple as possible that standard Python tooling should be used, meaning `pip` for installing the wrapper. Although calling this from `make` is relatively pain-free, it was found difficult to pass down all the options required to build and link against 2DECOMP&FFT, thus a rather unsatisfactory solution was developed to have the user hardcode these values into the `setup.py` script that forms part of the Python build system, this approach was chosen as a pragmatic solution as attempts to make this work had consumed a significant amount of effort. A new test was developed to populate a grid with global indices in the X pencil and transpose this from X→Y→Z and back again, testing the values at each step to confirm the transpose operations were performing correctly.

The parallel execution of Py4Incompact3D then follows that of Xcompact3D, the main effort required was to port the derivative implementation to work in parallel, the pencil decomposition makes parallelism in any given direction trivial, however the data must be appropriately transposed for the derivative. To keep a simple interface from the user's perspective all data in Py4Incompact3D is held in X pencils, calling the derivative routine will then transpose the input as needed, compute the derivative agnostically of the direction of derivative, and transpose the output back to the X pencil. This also means that the code maintenance burden is minimised (*c.f.* Xcompact3D which implements derivative subroutines per-pencil), however it does inhibit reuse of already transposed data. The parallel performance of Py4Incompact3D was evaluated by computing the gradient (*i.e.* $\partial/\partial x, \partial/\partial y, \partial/\partial z$) of four fields, comparable to the computing

effort that might be required in a typical postprocessing job based on four primary flow variables u, v, w, p. Parallel speedup is presented in *fig.* 4 for problem sizes ranging from 129^3 to 1025^3 from 1 core up to 64 full ARCHER2 nodes (8192 cores), a larger problem size (4097^3) was tested, however ran into OOM issues at lower node counts and floating point errors on larger node counts that will require further investigation.

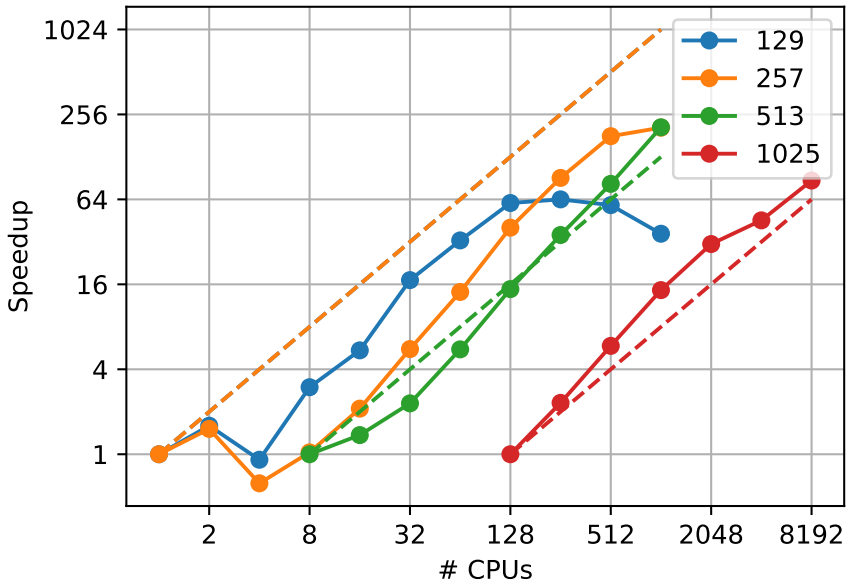

Figure 4: Speedup obtained by Py4Incompact3D computing gradients of four variables on problem sizes ranging from 129^3 to 1025^3, running on up to 64 ARCHER2 nodes (8192 cores). Dashed lines indicate the ideal scaling behaviour.

4 *In-situ* Postprocessing with Py4Incompact3D

As part of the parallelisation of Py4Incompact3D, the parallel readers from 2DECOMP&FFT were also added, enabling reading Xcompact3D output files in parallel, however this was found not to work well with ADIOS2, therefore Py4Incompact3D was interfaced directly with ADIOS2 which exposes a Python high level interface. Using this interface directly would have required a significant rewrite of Py4Incompact3D so that all processing was contained "within" ADIOS2 as the ADIOS2 Python interface is based on an `iterator` concept. When opening and closing a physical file this issue did not appear, however using ADIOS2 in the streaming mode, using the `SST` engine, would only work properly when using the iterator. By using the *advance* method of the engine object it was possible to hold the ADIOS2 "file" open in streaming mode allowing ADIOS2 to be used without requiring a full rewrite of Py4Incompact3D. Although this method is undocumented and thus might not be entirely safe to depend on, it appears to be part of the iterator mechanism and so seems a reasonable solution until a better one can be found. With this capability, an *in-situ* postprocessing script was written for the TGV problem, computing the enstrophy from the streamed velocity field and then integrating and taking the mean over the domain, for comparison the same computation which is currently performed *in-situ* by Xcompact3D was left in place and the output from each plotted in *fig.* 5. As can be seen, the evolution of enstrophy computed by Xcompact3D and Py4Incompact3D lie on top of each other, demonstrating that Py4Incompact3D can be a viable alternative to hardcoded analyses implemented in Xcompact3D.

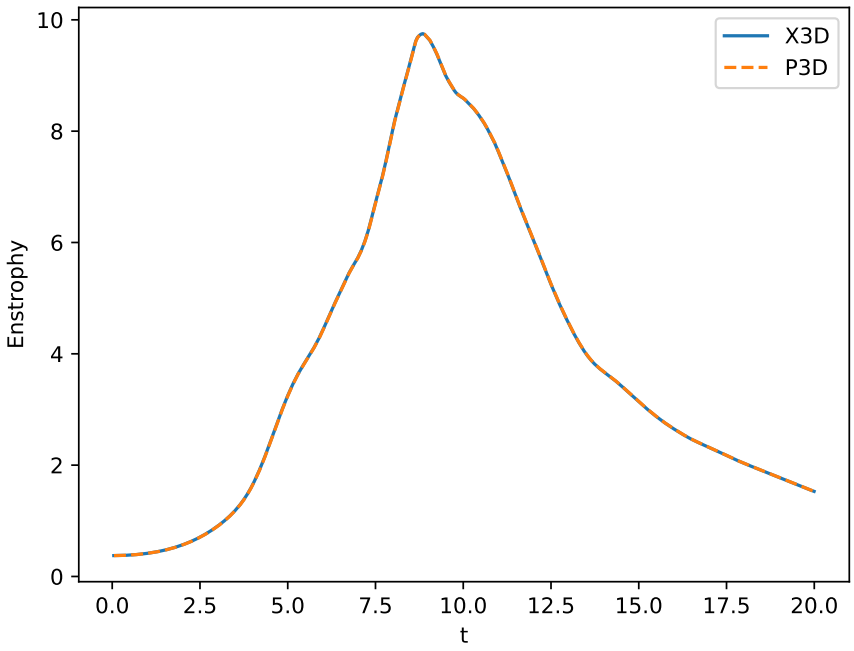

Figure 5: Comparing enstrophy evolution for the TGV case computed *in-situ* by Xcompact3D (`x3d`) and Py4Incompact3D (`p3d`), respectively.

4.1 Optimising the TriDiagonal Solver in Py4Incompact3D

One of the primary design goals of Py4Incompact3D was to enable analysis of Xcompact3D simulation results consistently with the numerical methods used to produce them. The compact finite difference schemes used for derivative approximation in Xcompact3D are of the form

$$\alpha f'_{i-1} + f'_f + \alpha f'_{i+1} = R(f)|_i \tag{1}$$

where f is the field, f' some computed function (first or second derivative, an interpolant), α and R a constant and function determined by the scheme, respectively, and subscript i indicates nodal location. Solution of (1) by the TriDiagonal Matrix Algorithm (TDMA) is efficient, and is the method used in both Xcompact3D and Py4Incompact3D, however in development and testing of the *in-situ* Py4Incompact3D application it was observed that the Py4Incompact3D processing was not able to keep pace with the stream of data from Xcompact3D, resulting in a backlog of data which would eventually cause an Out Of Memory error. This slowdown relative to Xcompact3D occurred only when the processing involved derivative evaluations, analyses which involved only field operations (such as evaluating and integrating kinetic energy over the domain) were able to keep pace, being purely implemented in `numpy` on the Python side. The derivatives are evaluated on a 3-D domain meaning there are two independent indices in any given direction, these loops are implemented in `numpy`, however due to the loop-carried dependency a Python loop is required for the main TDMA loop, as Python loops are known to be slow this was targeted for optimisation.

The TriDiagonal Matrix Algorithm consists of a forward elimination sweep followed by a backwards substitution sweep to compute the solution of the system. Although this appears directly translatable to `numpy` its design evaluates the right hand side of an expression before writing the result into the left hand side, as such, `numpy` cannot express loop-carried dependencies. As can be found at [7], these forward and backward sweeps take the form of a first-order inhomogeneous recurrence relation, for which there is an

explicit solution

$$
\begin{aligned}
x_{i+1} &= a_i x_i + r_i \\
\Rightarrow x_i &= \left(\prod_{j=0}^{i-1} a_j \right) \left(x_0 + \sum_{j=0}^{i-1} \frac{r_j}{\prod_{k=0}^{j} a_k} \right)
\end{aligned} \tag{2}
$$

This formulation expresses the solution as a combination of cumulative products and sums, operations implemented by `numpy`, allowing the TDMA solver to be expressed in `numpy` only. Testing revealed the division by a product of many floating-point numbers risks division by zero especially on larger grids, thus was implemented as

$$
x_i = \prod_{j=0}^{i-1} a_j x_0 + \sum_{j=0}^{n-1} \left(r_j \prod_{k=j+1}^{i-1} a_k \right) \tag{3}
$$

with the bracketed term taking the form of a matrix-vector product. With this optimised version in place local testing of the TGV case on a work station showed Py4Incompact3D keeping pace with Xcompact3d when run on 4 cores each.

Using this code the TGV case was run on ARCHER2 with the *in-situ* analysis performed either within Xcompact3D or using ADIOS2 and Py4Incompact3D on a 257^3 node mesh with timings shown in *fig.* 6.

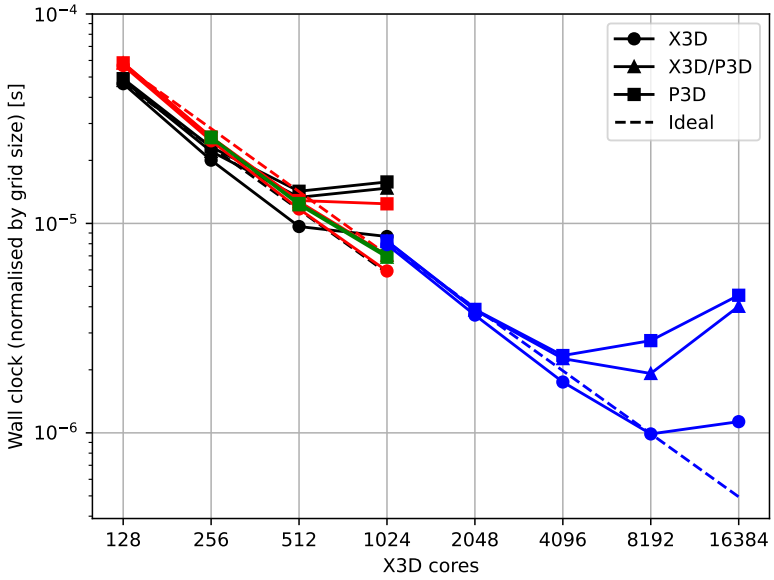

Figure 6: Timings for running the TGV case on a range of mesh sizes, normalised by mesh size. The x axis indicates the number of ARCHER2 cores Xcompact3D is running on. The markers as shown by the legend indicate Xcompact3d, Xcompact3d coupled to Py4Incompact3d and Py4Incompact3d timings, respectively, with dashed lines for ideal scaling of each problem. The problem is indicated by colour: 129^3 mesh (black) with 128 cores for Py4Incompact3d, 257^3 mesh with 128 cores for Py4Incompact3d (red), 257^3 mesh with 256 cores for Py4Incompact3d (green) and 513^3 with 1024 cores for Py4Incompact3d.

As *fig.* 6 shows, both Xcompact3D running standalone and Xcompact3D coupled to Py4Incompact3D show excellent scaling, up until an approximately 4:1 ratio of Xcompact3d to Py4Incompact3d cores across several problem sizes. The coupled setup clearly introduces an overhead when compared with the standalone

runs with an analysis of the timings in the Py4Incompact3D code showing that a significant fraction of time is spent in the data load function, that is either waiting for Xcompact3D to send the next data set or in the process of receiving it. This fraction ranges from a low of 20% when there is a very large 8:1 ratio of Xcompact3d:Py4Incompact3d nodes to over 80% of the Py4Incompact3d runtime when there is an equal division of resources.

5 Conclusion

An optional I/O backend was added to 2DECOMP&FFT/Xcompact3D using ADIOS2, yielding a performance benefit over the existing MPI-IO backend when writing to disk, the key difference between MPI-IO and ADIOS2 is believed to be maximising off-node bandwidth by writing a file per node in ADIOS2 *vs* one writer/aggregator per OST (maximum 12) in MPI-IO[2]. Following this, the Py4Incompact3D postprocessing tool was parallelised by wrapping 2DECOMP&FFT and strong scaling in line with what would be expected based on Xcompact3D's performance demonstrated up to 64 ARCHER2 nodes (8,192 MPI ranks) for problem sizes up to 1025^3 mesh nodes. Finally, *in-situ* processing was demonstrated with Py4Incompact3D and Xcompact3D, enabled at runtime through ADIOS2 configuration files. Testing on ARCHER2 demonstrated that this introduced only a small overhead and did not negatively impact the code scaling up to a 4:1 resource split.

With these new features it is now possible to move problem-specific analysis code out of the Xcompact3D code base and into Py4Incompact3D programs/scripts that will be easier to distribute and for users to modify. Although a start has been made, we were not able to demonstrate *in-situ* visualisation through combining Py4Incompact3D with Paraview Catalyst. As an example of possible further benefits this would bring, a 30 frame per second video of the TGV simulation at the scale studied by Dairay would require writing, then reading, ≈ 40 TiB of data $(20 \times 30) \times \left(2048^3 \times 8\text{Bytes}\right)$[3] to/from disk whereas rendering the scenes from an *in-situ* visualisation might write on the order of 100s of MiB.

Acknowledgements

This work was funded under the embedded CSR programme of the ARCHER2 UK National Supercomputing Service (https://www.archer2.ac.uk).

References

[1] Xcompact3D github. https://github.com/xcompact3d/Incompact3d. Accessed: 2022/10/04.

[2] Paul Bartholomew, Georgios Deskos, Ricardo AS Frantz, Felipe N Schuch, Eric Lamballais, and Sylvain Laizet. Xcompact3d: An open-source framework for solving turbulence problems on a cartesian mesh. *SoftwareX*, 12:100550, 2020.

[3] 2DECOMP&FFT github. https://github.com/xcompact3d/2decomp-fft. Accessed: 2022/10/04.

[4] ADIOS2 github. https://github.com/ornladios/ADIOS2. Accessed: 2022/10/04.

[5] The HDF5 library & file format. https://www.hdfgroup.org/solutions/hdf5/. Accessed: 2022/10/04.

[6] Py4Incompact3D github. https://github.com/xcompact3d/Py4Incompact3d. Accessed: 2022/10/04.

[7] Solving first-order non-homogeneous recurrence relations with variable coefficients. https://en.wikipedia.org/wiki/Recurrence_relation#Solving_first-order_non-homogeneous_recurrence_relations_with_variable_coefficients. Accessed: 2022/10/04.

[2] In discussions at EPCC the I/O bottleneck is thought not to be the disks themselves.

[3] Even saving output at single precision would only reduce this data volume half.