eCSE07-15: Optimizing the I/O performance of OpenFOAM for massively parallel high-fidelity CFD simulations

Stefano Salvini, Neil Ashton, Ian Bush, Wes Armour

March 2, 2018

Abstract

In this report we detail the work undertaken to develop a parallel I/O solution for OpenFOAM. We include an analysis of the current state of the code, the improvements we have made and finally some benchmark results.

This work was funded under the embedded CSE programme of the ARCHER UK National Supercomputing Service (http://www.archer.ac.uk)

1 Introduction and Project Description

The aim of this project was to develop a *prototype* which introduced modern, parallel, large scale IO into OpenFOAM [1] (http://www.openfoam.org), thereafter referred to as OF. We chose HDF5 [2] as it is widely used in academic as well as industrial applications.

The large OF user base and their need for continuity,implied that we needed to propose the following requirements

1. Minimum modifications to the code, no changes in users' interface, data, etc.
2. Default behavior exactly as standard OF with optional runtime choice of OF files or HDF5.
3. Reduce the number of output files, while maintaining the ability to remove cleanly time dumps already stored (for example, time dumps outside a "storage window"), this is a facility in OF which we intended to preserve.
4. Simplify users' editing in the case of restart.
5. Clean compilation using the standard OF mechanisms.
6. Testing on small parallel system.
7. Testing and benchmarking on large systems.
8. Creating a comprehensive **GitHub** repository, with sources, instructions and information. The repository can be accessed through github: https://github.com/stefsal/OeRC_OpenFOAM_HDF5

All have been achieved with the total number of files per checkpoint being reduced from the number of processors× number of variables to either a single file or multiple files if processor bunching is used. The following sections describe the modifications to the code and the results of the benchmarking exercise.

2 Methodology

OF (OpenFOAM) is a comprehensive, flexible, widely used package for CFD computation. Its flexibility allows users to plan, implement and solve a wide range of problems problems using "semi-symbolical" definitions. Items such as discretization and solution methods can be defined transparently, avoiding the finer details of their implementation. A range of templates, solutions, codes etc. are also available.

Whilst OpenFOAM has many strengths it's code design has resulted in some fundamental weaknesses for I/O.

1. Parallelism is not optimal. OF uses MPI, but MPI is linked internally *within* the executable at em run time, rather than at link time. That implies that MPI facilities in use and their locations in the code are limited, a sort of MPI "dummy" library is for serial use. The intention was to have a single executable. However, irrespective of whether OF is called through `mpirun`, as MPI is linked in the executable (source level), the execution parameter "`-parallel`".

2. The point above implies that widely used, third party, advanced parallel libraries such as PETSci (ANL) cannot be currently used.

3. The OpenFOAM I/O system has several characteristics that make it less than ideal for HPC systems

   (a) At each of N_T time dumps (steps) of interest, each of the N_P processes creates and writes a file for each of the N_F quantities required (e.g., pressure, velocity field, etc.). Thus the number of files created is $F = N_t \cdot N_P \cdot N_F$. For example, a large-ish scale problem, with $N_T = 100$, $N_P = 10000$, $N_F = 20$, would generate 20 million files. This would be, to say the least, impractical in a non-dedicated large-scale HPC environment.

   (b) Objects types, such as scalars, vectors, tensors, matrices etc. are defined in OF, but only accessible at IO time as collections of individual data items, stripped of any metadata. For examples, when writing a vector, the length of the vector is not available.

   (c) Each individual entry (number) is output independently directly to an external file. This is very inefficient as no pipelining is possible and use of full cache lines is denied, thus increasing memory traffic, etc.

   (d) The code is extremely complex. For example, attaching metadata to the objects values, would be costly and could cause unpredictable side-effects without in-depth changes to the code.

   (e) The mechanism for input data is cumbersome, where each element is read in character by character then decoded.

4. Restarts pose a particular problem. To edit/change the boundaries, files are consolidated across all processes in a single file. The interior values, the major portion, are at the top, the boundary values, the smallest portion at the bottom. As no tools are provided, users need to manually edit the consolidated file using standard editors: for large problems can be very large (possibly 100s GBytes). Users' have reported that this could take a very long time, if at all practical.

Given initial unavoidable staffing issues, which led to some lost time, we decided to focus on several achievable targets:

1. Input from files would not be considered, because of its complexity, the resources it would take and the potential deep modifications of the code. Reluctantly, we decided that that would also apply to initial data, such as distributed cells data etc. These files are also fewer, in most cases, than output files. Future work is however to deliver this functionality.

2. We chose to use HDF5. HDF5 is used by a great number of academic, industrial and commercial large scale applications in many different fields and locations.

We proposed the following requirements

1. Minimum modifications to the code, no changes in users' interface, data, etc.

2. Default behavior exactly as standard OF with optional runtime choice of OF files or HDF5.

3. Reduce the number of output files, while maintaining the ability to remove cleanly time dumps already stored (for example, time dumps outside a "storage window"), this is a facility in OF which we intended to preserve.

4. Simplify users' editing in the case of restart.

5. Clean compilation using the standard OF mechanisms.

6. Testing on small parallel system.

7. Testing and benchmarking on large systems.

8. Creating a comprehensive **GitHub** repository, with sources, instructions and information. The repository can be accessed through github: https://github.com/stefsal/OeRC_OpenFOAM_HDF5

All requirements have been implemented in full:

1. Achieved. Only 6 (six) sources have been modified: (OFstream.C , OFstream.H, OSstream.C, OSTreamI.C , OSstream.H and regIOobjectWrite.C)

2. Achieved. Notice that because of code complexity and in order to avoid extensive modifications, potentially involving a large number of compilation units, two environment variables have been created to select (details in github):

- Using OF standard or HDF5 files.
- 'Bunching' output from processes (see below) to maximize system throughput.

3. Achieved. Either of two modes of operation can be selected:

   All-processes one HDF5 file is written by all processes for each time stamp for a total number of N_T files. In the example above, 100 files would be created, not 10,000,000.

   Bunching Processes are grouped into bunches, of size B to create N_P/B HDF5 files: if in the example, $B = 100$, then 100*100 = 10,000 HDF5 files would be created.

4. Achieved. Interior and boundary values are now stored separately in HDF5 files: they can be concatenated trivially to form the original OF file.

5. Achieved: replace the six sources with those in the github repository, modify two compiler/linker options file then build OF using the standard scripts/commands. Full instructions are included in the github repository.

6. Achieved: Extensive testing has been carried out.

7. Achieved: Benchmarking has been completed on ARCHER.

8. Achieved: https://github.com/stefsal/OeRC_OpenFOAM_HDF5 .

2.1 HDF5 implementation

To preserve the possibility of deleting individual time dumps, for example retaining only the last few, we needed a suitable structure. In OF, each process create a separate directory, indexed by process number; there is then a subdirectory for each time dump.
We had two issues in creating a suitable structure:

1. using the process name as an identifier (one HDF5 file for each process) would create N_P HDF5 files, with appropriate structures within, very much like in OF directories. However, deleting time dumps would not be straightforward: space would not be recovered, just simply lost. The only way to recover it would be to "recompact" the HDF5 file, which is rather time consuming.

2. The alternative would be storing within each process directory a separate HDF5 file for each time dump. This, unfortunately, would reduce the number of files only by a factor N_F to $N_T * N_P$. In our example, $100 \cdot 10000 = 1,000,000$ files would still be created.

- Hence we *inverted process name and time dump* in the storage structure: there is an HDF5 for each time dump, and within it all data (datasets) are grouped by process name. In other words, for each process name, the data structure is the same. That allows the clean removal of time dumps, without incurring in the issues at the previous point.

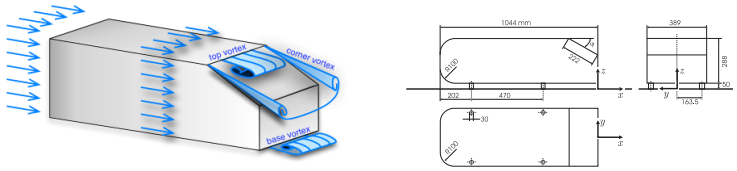

Figure 1: (a) Ahmed car body flow physics and (b) Ahmed car body dimensions

- The HDF5 dataset (*files*contents are byte-wise identical to the OF standard files. Extracting them shows exactly that.
- Using HDF5 does not write the standard OF files and then copy them. Output is instead written to an internal buffer then streamed in bulk to HDF5. At no time, unless no HDF5 is not required, are standard OF files created or written. HDF5 files are written directly from within OF.
- Time constraints prevented us to implement full MPI-parallel HDF5. MPI-based HDF5 APIs were only available for C and not C++, so all interfaces to HDF5 would need to be reworked, including any OF data required by them.
- Currently, we use HDF5 serial in two different modes, as previously mentioned:

  All-processes one HDF5 file is written by all processes for each time stamp for a total number of N_T files. In the example above, 100 files would be created, not 20,000,000. As the HDF5 files can only be written *serially*, a parallel *lock* allows only one process at the time to write the data to HDF5 file. However, the data have been written to a buffer beforehand, hence the whole buffer is written in one go, fully using any pipelining that the system can offer.

  Bunching Processes are grouped into bunches, of size B to create N_P/B HDF5 files: if in the example, $B = 100$, then 100*100 = 10,000 HDF5 files would be created.
  there is one independent lock per bunch, so bunches can write *disjoint* HDF5 files independently and concurrently. HDF5 file names now show not just the time but the lowest and highest process number in the bunch. The network bandwidth can therefore be better exploited. If b_P is the bandwidth from each process to storage, the total bandwidth used would be $b_B = \max(b_P \cdot N_P/B, b_N)$, where B_N is the overall network bandwidth.

3 Benchmarking

To benchmark the code, the Ahmed car body test-case was chosen. The Ahmed car body [3][4] represents a generic car geometry with a slanted back and a flat front and has been extensively tested in the literature [5].

3.1 Computational grid and boundary conditions

The flow is at a Reynolds number of $Re = 768,000$ based on the body height and the free-stream velocity $U_\infty = 40ms^{-1}$. An inlet condition is imposed at upstream of the body and an outlet condition is imposed downstream. A no-slip wall condition is imposed on the ground floor and car body, with slip conditions applied to the wind tunnel walls.

An unstructured snappyHexMesh generated 13 million cell mesh was used - for this mesh, the first near-wall cells over the car body had a $y^+ < 1$ and the refinement was concentrated on the near-wall and separation regions. The time-step is 0.0002s and the simulation is run for 2s of physical flow-time resulting in 10,000 iterations.

3.2 Results

The test-case was simulated on ARCHER using the cray-hdf5/1.8.14 HDF5 module and a custom compilation of OpenFOAM 4.1 with the only modifications being the 6 six sources files mentioned previously. Benchmarking simulations were conducted on 192 cores (67,000 cells per core) although similar trends were observed for larger core counts. For each 10 checkpoints were taken but the timing data is for a single checkpoint. Table 1 shows a breakdown of the time for each checkpoint together with the number of files.

	Time for checkpoint	Files	Zip one checkpoint
OF Default	1s	38,400	600s
HDF5 (No bunching)	152s	10	
HDF5 (Bunches=40)	21s	50	
HDF5 (Bunches=20)	6.5s	100	
HDF5 (Bunches=10)	3.6s	200	140s
HDF5 (Bunches=2)	3.5s	1000	

Table 1: Time and number of files for the checkpoints from different I/O methods

It can be seen that the HDF5 timing is strongly correlated to the number of bunches, which is due to the network bandwidth explained in the previous methodology section. The optimum number of bunches is 10-20 depending on the desired number of files. All HDF5 solutions produce large reductions in the number of files, which has the associated advantage of using usability. It is common to transfer or compress checkpoints after simulations. Working with fewer files means these times are massively reduced, which outweighs the increase in the checkpoint during the simulation. Typical simulations don't checkpoint more than 10 times so the time to checkpoint is very small compared to the entire simulation. For the simulation with 192 cores, the total simulation time to 2s of physical time was 27hrs, whereas checkpoint time was only 36s for HDF5 (Bunches=10).

4 Conclusions

This project has delivered a HDF5 based parallel I/O solution for OpenFOAM. This new system reduces the number of files by a factor proportional to the number of processors e.g a 192 core simulation with 10 checkpoints from 38,400 files to just 200. This provides a huge improvement in the usability of the code for very small increase in total runtime. Whilst the code is fully functional, further work described below will be pursued to deliver greater improvements.

All codes and instructions are available at `https:github.com/stefsal/OeRC_OpenFOAM_HDF5`

4.1 Future work

Some further developments would be:

- Full MPI HDF5 implementation. This would require using C rather than C++ APIs (OF is written in C++).
- Updating completely both input and output to allow easy access to HDF5 files.
- Developments to allow the use of external high quality libraries such as the previously mentioned PETSci.
- Tools for quickly editing boundary values files (datasets) for restart.
- Better mechanisms for sleeker and parallel post-processing, accessing HDF5 without consolidating files.

References

[1] Hrvoje. Jasak. *Error analysis and estimation for the finite volume method with applications to fluid flows.* PhD thesis, 1996.

[2] The HDF Group. Hierarchical Data Format, version 5, 1997.

[3] S. R Ahmed, G Ramm, and G Faltin. Some salient features of the time averaged ground vehicle wake. *SAE-Paper 840300*, 1984.

[4] H Lienhart and S Becker. Flow and turbulent structure in the wake of a simplified car model. *SAE*, 01(1):0656, 2003.

[5] N Ashton and A Revell. Key factors in the use of DDES for the flow around a simplified car. *International Journal of Heat and Fluid Flow*, 54:236–249, 2015.