Technical Report for eCSE 10-2: GeoChemFOAM on ARCHER2

Dr Gavin Pringle, EPCC; Dr Julien Maes, HWU; Dr Hannah P Menke, HWU

18 March, 2025.

Executive Summary

The eCSE 10-2 project, "GeoChemFOAM on ARCHER2," has significantly enhanced the capabilities available to users by making GCF-v5.1 and a 64-bit version of OpenFOAM centrally available on ARCHER2. Comprehensive user documentation is provided on the GCF wiki, facilitating easy access and installation for all ARCHER2 users.

The project successfully installed and compiled GCF-5.0 and GCF-v5.1 on ARCHER2, resolving several parse and compiler errors. A Python virtual environment was created to support the necessary modules. Additionally, a 64-bit version of OpenFOAM was installed to handle very large meshes, enabling larger mesh sizes than previously possible. GCF-5.0 was made publicly available via ARCHER2 modules, with comprehensive user documentation provided on the GCF wiki.

The project faced significant challenges in adapting the inherently serial pre-processing tools in OpenFOAM, specifically blockMesh and decomposePar. These tools must loop over the entire mesh and manage I/O operations, making them difficult to parallelize. Various approaches, such as using existing parallel mesh generators or virtual memory, were considered but found to be complex and potentially inefficient. Profiling these tools revealed significant variations in execution times due to I/O operations, further complicating the parallelization efforts. The extensive and intricate source code required careful modification to remove unnecessary computations while ensuring correct output, making the process labour-intensive and reliant on trial-and-error.

Given the challenges with parallelizing blockMesh and decomposePar, the team developed a custom Python routine, createMesh.py. This routine leveraged the simplicity of cubic cells and a cuboid mesh, allowing for straightforward division and distribution across multiple processors. Initially designed as a serial routine, createMesh.py was later parallelized using mpi4py, enabling concurrent creation of processor directories and their associated polyMesh files without writing a global polyMesh directory.

Profiling efforts revealed that memory usage and execution times varied wildly in the shared environment of ARCHER2. Despite these inconsistent results, key findings indicated that createMesh.py is I/O-bound, with performance heavily dependent on the I/O subsystem's load. The project demonstrated that createMesh.py enables the generation of previously prohibitive mesh sizes, regardless of efficiency considerations.

Several strategies were implemented to reduce the memory footprint and improve the performance and scaling of createMesh.py. These included inlining subprocesses, removing unnecessary arrays, preloading Python libraries, optimizing loop invariants, introducing

directional padding, and modifying data reading to a layer-by-layer approach. These optimizations significantly improved performance and parallel efficiency, particularly for larger core counts.

The resultant parallel efficiencies of createMesh.py were significant, particularly for larger core counts. Scaling tests within a single node, conducted on a 400^3 mesh, demonstrated parallel efficiencies of 98% on 16 cores, 94% on 30 cores, and 63% on 128 cores after optimizations. These improvements highlight the effectiveness of the implemented optimizations in enhancing the script's scalability and performance.

Running ParaView to visualise results on ARCHER2 involved overcoming several challenges related to memory usage and performance. By testing different configurations and closely monitoring memory usage, the team was able to optimize the process and derive an equation to estimate the number of nodes required for different mesh sizes.

The eCSE 10-2 project has enabled vastly larger meshes by making the memory of multiple nodes instantly available, facilitating simulations on larger datasets. This advancement leads to more sophisticated research opportunities for GeoChemFoam users and the broader OpenFOAM community, especially those on the road to Exascale computing.

Introduction

The “GeoChemFOAM on ARCHER2” project was conceived to significantly advance the computational capabilities of GeoChemFoam—a state-of-the-art open-source pore-scale physics numerical solver—by harnessing the power of high-performance computing on ARCHER2. GeoChemFoam is pivotal in simulating complex processes such as reactive transport, multiphase flow, and heat transfer in porous media, which are crucial for advancing clean energy technologies including carbon capture and storage, hydrogen storage, geothermal energy, and low-carbon building materials.

Installation

Installation of GCF-5.0

Installation and Compilation: both installed in a personal account on ARCHER2 project space. Encountered and resolved several parse errors and a compiler error. Third Party Warnings: Warnings in Third Party code left unfixed but fixed in main code. Further, we fixed a bug in Third Party code locally. A compiler error was encountered in the ThirdParty/Phreeqc/src/RM_interface_F.cpp file. The error was due to a pointer being incorrectly referred to by "I1" rather than by "*I1".

Python Virtual Environment: Created and configured a Python virtual environment for running tests. The Python Virtual Environment required modules that had changed names, such as "stl" to "numpy-stl", along with these necessary modules: matplotlib, scikit-image, numpy-stl, and h5py.

We made GCF version 5.0 publicly available via ARCHER2 modules. Users can load the GCF-5.0 module on ARCHER2 to access the software and its functionalities without needing to install it individually.

User Documentation is provided on how to use the module version or how users can install it on their own Archer2 file space, are available via the GCF wiki: https://github.com/GeoChemFoam/GeoChemFoam/wiki/GeoChemFoam-on-ARCHER2.
This wiki includes detailed steps for installing GCF-5.0 on ARCHER2, including resolving common issues such as parse errors and compiler errors. There are also instructions on configuring the environment, including setting up the necessary modules and converting DOS files to UNIX format, and guidelines on running tests and tutorials using GCF-5.0, including creating and using Python virtual environments.

The hpc-uk github public repository, also includes instructions to allow all Archer2 staff to support the central installation if the owner is unavailable. This contains examples of batch scripts for running GCF-5.0 on ARCHER2, including configurations for parallel execution and SLURM compatibility. There is also documentation on the bug fixes applied during the installation process, including specific changes made to the Third Party code and the main codebase, and details on the profiling and optimization efforts, including improvements made to the createMesh.sh script and other routines.

Installation of OpenFOAM 64-bit

A 64-bit version of OpenFOAM was also made centrally available on Archer2. The project involved creating and processing very large meshes, which required OpenFOAM to run with 64-bit addressing. It has been built by simply adding `-WM_LABEL_SIZE=64` to the configuration files. While the full installation took around three hours, it provided a slight performance improvement, a negligible increase in the executables' memory footprints, but enabled far larger mesh sizes than previously viable. To access this, use: `module load other-software; module load openfoam/v2212_64`

Installation of GCF-v5.1

GCF-v.51 was also installed on Archer2 in both 32-bit and 64-bit versions, where the 32-bit is available centrally via modules. The instructions on how to install this on Archer2 centrally are given in the uk-hpc git repository, and instructions for Archer2 users to install it locally in their own file space are given in the GCF wiki.

Parallelising OpenFOAM pre-processing tools

OpenFOAM’s pre- and post-processing tools are serial and, as such, our meshes are limited by the memory of one node. However, now that GeoChemFoam is on ARCHER2 users can exploit the full 256GB of ARCHER2’s regular nodes or even the 512GB of the larger nodes.

The rest of the eCSE worked to parallelise two previously ‘inherently serial’ routines in OpenFOAM: the meshing tool blockMesh, and the partitioning tool decomposePar.

Parallelization of blockMesh and decomposePar

The team started by copying the source code of blockMesh and creating a private version for development. They removed unnecessary code and routines to streamline the tool for their specific use case. The goal was to create a version of blockMesh that could run concurrently, creating processor directories and their associated polyMesh files without writing a global polyMesh directory.

Similar to blockMesh, the team created a private version of decomposePar and removed unrequired source code. They aimed to join blockMesh and decomposePar into a single routine that could run in parallel, avoiding the need for serial polyMesh I/O.

Challenges with Parallelizing blockMesh and decomposePar

These tools must loop over the entire mesh and manage I/O operations, making it inherently serial. Various approaches to parallelize it, such as using existing parallel mesh generators or virtual memory, were considered but found to be complex and potentially inefficient.

Profiling these tools revealed significant variations in execution times due to I/O operations, which are shared resources. This made it difficult to achieve consistent performance improvements. Further, different processor layouts and filesystems (LFS vs. SSD) were tested, but the results were inconsistent, further complicating the parallelization efforts.

The source code for blockMesh and decomposePar is extensive and tricky to modify, as removing unnecessary computations and routines while ensuring the tools still produced the correct output was a challenging task. The team had to rely on trial-and-error and manual inspection of user guides, wikis, and source code to identify and remove unnecessary routines. Finally, profiling tools like Cray-pat-lite and gprof were tried but did not provide the necessary insights, making the process more trial-and-error.

Alternative to blockMesh&decomposePar: createMesh.py

Given these challenges, the team decided to write their own Python code, createMesh.py. Writing createMesh.py from scratch allowed the team to have complete control over the code, making it easier to implement and test specific optimizations.

Writing our own custom-built, manually crafted Python routine was feasible due to several specific characteristics and constraints of the problem:

Cubic Cells: The mesh cells were cubic, which simplified the geometry, and the calculations required for mesh generation. Cubic cells have uniform dimensions, making it easier to handle and manipulate them programmatically.

Cuboid Mesh: The overall mesh was a cuboid, meaning it had a regular, rectangular shape. This regularity allowed for straightforward division and distribution of the mesh across multiple processors.

Simple Mesh Structure: The local mesh was simpler than the more complex meshes that blockMesh typically handles. This simplicity meant that we could avoid the additional complexity and overhead associated with more intricate mesh structures.

No Grading: The mesh did not require grading, which is the process of gradually changing the size of the cells. Without grading, the mesh generation process was more straightforward, as all cells were of uniform size.

Single Block with No Grading: Each processor handled just one block with no grading. This uniformity across processors made it easier to implement a parallel routine where each processor could independently generate its portion of the mesh.

Refinements: the absence of refinements in the current implementation significantly simplified the development of our custom-built, manually crafted Python routine.

Avoiding Unnecessary Computation: By focusing on cubic cells and a cuboid mesh, we could remove unnecessary computations that would be required for more complex geometries. This reduction in complexity made the custom routine more efficient.

Understanding of Mesh File Structure: The team had a good understanding of the structure of the mesh files generated by blockMesh. This understanding allowed them to recreate the necessary files programmatically, ensuring compatibility with the existing workflow.

Parallelization: The custom routine was designed to run concurrently, with each processor creating its own portion of the mesh. This parallel approach avoided the need for sequential writing and reading of large mesh files, improving performance. The initial version of createMesh.py was designed to be serial, producing the same output as running blockMesh followed by decomposePar. Once the serial version was validated, the team parallelized createMesh.py using mpi4py, a Python package for parallel computing with MPI. The parallelized createMesh.py was designed to run concurrently, creating processor directories and their associated polyMesh files without writing a global polyMesh directory.

Initial Performance

Investigated memory usage and runtime performance to and optimized code to reduce memory footprint, runtime performance

Tools used to measure memory usage:

- Sstat:
    - Used to display memory usage of running jobs.
    - Key metrics included:
        - AveRSS: Average memory use per process over the job's duration.
        - MaxRSS: Maximum memory use by an individual process during the job.
        - MaxRSSTask: Process ID associated with the maximum memory use. ■ MaxRSSNode: Node ID associated with the maximum memory use.
        - TRESUsageInTot: Totals of various properties for the job, including total memory use.

- archer2jobload:
    - Provided a cleaner display of CPU and memory usage for running jobs.
    - Could auto-refresh every few seconds to monitor real-time usage.
- sacct:
    - Displayed accounting data and memory usage for finished jobs.
    - Provided detailed information on memory usage, including average and maximum values.
- perf-report:
    - Initially looked promising but reported less RAM than the image file alone.
    - Measured each processor's memory footprint, more suited to C, C++, and Fortran, with basic support for Python.

The tools used for memory and performance analysis produced unreliable, inconsistent, and sometimes ineffective results due to several factors:

Variability in Execution Times: Execution times varied wildly due to significant I/O operations, which are shared resources on the system. This variability made it difficult to draw consistent conclusions from the results.

Inconsistent Memory Usage Reports: Tools like sacct and perf-report provided inconsistent memory usage reports. sacct was too noisy to provide clear insights, while perf-report reported less RAM usage than the image file alone, which was clearly incorrect.
Impact of I/O Operations: The performance of the tools was heavily influenced by I/O operations. For instance, perf-report showed a greater range of minimum and maximum times than unix time, confirming that I/O time dominated and used shared resources. The assumption that perf-report would be more consistent was false, as times varied more widely than with unix time.

Tool Limitations: sacct and perf-report had limitations in accurately measuring memory usage and execution times for Python scripts. perf-report was more suited to C, C++, and Fortran, with only basic support for Python. These tools did not account for the overhead of loading and unloading Python libraries, which added significant variability to the results.

Parallel Efficiency Issues: Some routines, like createcellmesh.py, had extremely poor parallel efficiency, which dragged down overall performance. This was partly due to the lack of I/O operations in single core runs, leading to misleading efficiency metrics. However, the overriding influence on performance was affected by the order in which routines were run, as the first routine incurring a 5-second hit due to the overhead of establishing core connections to the filesystem.

Unreliable Timing Tools: Cray-pat-lite and gprof, were ineffective, with both failing to generate any output files for analysis. Moreover, createMesh.py employs subprocesses which cannot be profiled without instrumenting the code.

Tools summary

Despite unreliable memory reporting of memory use and execution times, where both would vary wildly between instances of the same run, we were able to determine that

a) memory usage was dominated by system and Python libraries rather than the code itself. Impact of reading large image data was minimal, with most memory usage coming from other elements.
b) createMesh.py employs subprocesses which cannot be profiled without instrumenting the code.

The clear result was that createMesh.py is purely I/O bound and, as such, performance depends on how busy the I/O subsystem is. Regardless of efficiency considerations, createMesh.py facilitates the generation of meshes of previously prohibitive sizes.

Optimisations

To reduce the memory footprint and to improve performance and scaling of the createMesh.py Python script, several strategies were implemented and tested. Here are the detailed steps and techniques used:

Inlining Subprocesses: The team aimed to reduce the overhead associated with creating and destroying Python subprocesses, each of which has its own memory space. By inlining the five subprocesses into a single script, the team hoped to run the code faster and reduce RAM usage. However, tools reported increased RAM usage and unaffected execution times for low core counts, indicating that this approach did not yield the expected benefits.

Removing Unnecessary Arrays: The img and img_crop tables/arrays were removed from the script to reduce RAM usage. Although this theoretically reduced RAM, the tools did not report any significant effect on memory usage.

Preloading Python Libraries: Preloading all necessary Python libraries before running the main script helped reduce the time spent loading libraries during execution. This approach was particularly beneficial for low core counts, where the overhead of loading libraries could be significant. For higher core counts, the benefits were less pronounced, but overall runtime and parallel efficiency improved.

Optimizing Loop Invariants: The team examined the createcellmesh.py script and removed the calculation of loop invariants from within the `f.write` statement. By precomputing these invariants outside the loop, the script's runtime was reduced, and the overall negative impact on performance was minimized. This optimization led to lower computation-to-communication ratios, which slightly reduced parallel efficiency for this routine but improved overall performance.

Directional Padding: In the createEpspar.py script, directional padding was introduced to optimize the code. Instead of padding all directions and then cropping, the script now pads only the necessary directions, reducing the amount of data processed and stored. This optimization helped reduce the memory footprint and improved performance.

Layer-by-Layer Data Reading: The createEpspar.py script was modified to read data layer-by-layer instead of reading the entire image file at once. This change allowed the script to handle larger datasets without exceeding the memory limits of a single node. The performance of the new approach was comparable to the old method, with minimal impact on execution times.

Avoiding Unnecessary Preprocessing: To further reduce computation, the script was modified to delay reshaping, cropping, and padding the local array until after ownership of the data was confirmed. This approach ensured that only the necessary data was processed, reducing the overall memory footprint.

Scaling Results for createMesh.py

The scaling tests were conducted using different problem sizes, primarily focusing on a 400^3 image and a 1000^3 image. These tests aimed to evaluate the performance and parallel efficiency of the createMesh.py script across various core counts.

Results for 400^3 Image Scaling: Core Counts Tested: 2, 4, 8, 16, 30, and 128 cores. Parallel Efficiencies: 2 cores: 100% (baseline); 16 cores: 98%; 30 cores: 94%; 128 cores: 63% (improved from 49% after optimizations)

The final version of createMesh.py demonstrated significant improvements in performance and parallel efficiency, particularly for larger core counts. The optimizations implemented, such as writing data in one go, preloading Python libraries, and removing loop invariants, were effective in enhancing the script's scalability. The detailed component-wise analysis provided insights into which parts of the script required further optimization, guiding future development efforts.

Miscellaneous

Mesh Generation and Refinement

An investigation was started, comparing OpenFOAM Static- vs Dynamic- Mesh Refinement: Discussed methods for static mesh refinement using OpenFOAM utilities.

In OpenFOAM, static mesh refinement involves subdividing existing cells into smaller cells to achieve finer resolution in specific regions. Cells are initially numbered sequentially starting from 0 when the mesh is generated using blockMesh. Refinement regions are defined in the system/refineMeshDict file, specifying which cells or regions to refine. Each parent cell is subdivided into smaller child cells based on the refinement level (e.g., a 3D cell at level 1 is divided into 8 smaller cells). Numbering of Refined Cells: Original parent cells are no longer listed or referenced after refinement, and new child cells are assigned numbers starting from the next available index after the last original cell, ensuring consistent and sequential numbering. If multiple cells are refined, the numbering continues sequentially, ensuring unique identifiers for all cells. The polyMesh directory is updated with new faces and points definitions to reflect the refined cells.

In parallel simulations, numbering is local to each processor, with new cells numbered within their respective subdomains. Inter-processor communication ensures correct management of boundary conditions and data dependencies. This systematic approach maintains a consistent and organized mesh structure, ensuring compatibility with solvers and other utilities used in the simulation process.

Interesting to note that, while the OpenFOAM documentation provides a foundation, the detailed understanding of cell numbering for static mesh refinement must be gleaned from practical experimentation, reviewing source code, and leveraging community knowledge. This hands-on approach allows users to uncover the specific details and nuances that are not explicitly covered in the official documentation.

Paraview and Visualization

Paraview on ARCHER2:

Initial attempts to run ParaView on ARCHER2 encountered various errors, indicating that the process was tricky and required troubleshooting. One of the significant challenges was managing memory usage. ParaView often ran out of memory, especially when dealing with large datasets.

Switching to ParaView 5.13 helped address some of the initial issues. Ensuring that the 64-bit version was used was crucial for handling large datasets.

Various node and core counts were tested to optimize memory usage and performance: Running ParaView on a single node with varying core counts (e.g., 64 cores, 16 cores) to balance memory usage and execution time; distributing the workload across multiple nodes (e.g., 2, 4, 8 nodes) to manage memory better and improve performance.

Detailed observations were made regarding system memory usage and the memory used by ParaView: On a single node, memory usage varied with core count, with higher core counts using a significant portion of the node's memory. Distributing the workload across multiple nodes helped manage memory usage more effectively, with system memory usage and ParaView memory usage being monitored closely.

Execution Time and Cost: Execution times were measured, and the cost of running ParaView was evaluated based on different configurations. It was found the cheapest and most efficient configuration was identified as using 64 cores per node

To determine a rough guide, an equation was derived to determine the number of nodes required for a given mesh size based on memory usage observations:

- EstailladesCubeSmall: For a mesh with dimensions 572x572x3000 cells (approximately 1G cells), 4 nodes were required to use about 50% of the node's memory.
- EstailladesCube: For a larger mesh with dimensions 1144x1144x6000 cells (approximately 8G cells), 16 nodes were required to use about 50% of the node's memory.
- BentheimerCube: For an even larger mesh with dimensions 1950x1950x10800 cells (approximately 42G cells), 92-180 nodes were estimated to be required.

From this, a rule of thumb is the number of Archer2 nodes is the ratio of the Mesh size in G cells over the memory per node in TiB, where each Archer2 node has 256 GiB of memory.

Running ParaView on ARCHER2 involved overcoming several challenges related to memory usage and performance. By testing different configurations and closely monitoring memory

usage, the team was able to optimize the process and derive an equation to estimate the number of nodes required for different mesh sizes. This approach ensured efficient use of resources and improved the overall performance of ParaView on ARCHER2.

Paraview on Ultra2

Ultra2 is a high-performance computing (HPC) resource located at the Edinburgh International Data Facility (EIDF). Ultra2 is a single node, with 576 cores, and 18 TiB of memory. A new interactive queue enables us to run ParaView interactively. Moreover, a recent exodus of users means we are now permitted to employ 75% of the memory, but only now and then. Normal usage would have a 50% upper limit; however, this is still equivalent to 36 Archer2 nodes and interactive use of these nodes would be unrealistic.

Overall Impact

The project has successfully delivered a significantly enhanced meshing capability for GeoChemFoam on ARCHER2, including the installation of GCF-v5.1 and a 64-bit version of OpenFOAM, thus enabling simulations on much larger datasets. While some components (WP2 and WP3) remain as future work, the achievements in WP1 have already paved the way for improved high-performance computing capabilities for porous media research.

Outputs and Dissemination

The project has generated several high-impact outputs and extensive dissemination activities that underscore its success and broader influence within the high-performance computing and porous media simulation communities:

Software Modules on ARCHER2:

GCF 5.1 Module: The updated GeoChemFoam 5.1 module has been successfully compiled and made publicly available on ARCHER2. This enhanced version integrates the advancements achieved during the project, providing users with a more robust and scalable simulation tool.

OpenFOAM 64-bit Module: In addition, a 64-bit version of OpenFOAM has been deployed on ARCHER2, enabling the handling of significantly larger mesh sizes. This module facilitates advanced simulations by overcoming previous memory limitations associated with 32-bit systems.

Scholarly Publications:

A manuscript detailing the development, challenges, and performance improvements of the parallel meshing routines is currently in preparation for submission to a peer-reviewed journal. This paper will provide an in-depth account of the technical innovations and their impact on pore-scale simulations.

Conference and Poster Presentations:

Our team actively participated in the 2024 ARCHER2 Celebration of Science, where we presented a poster that showcased the key advancements made in GeoChemFoam and its integration on ARCHER2. We are also planning to attend and contribute to the 2025 ARCHER2 event, further engaging with the HPC community and sharing our latest findings.

Media and Outreach:

An EPCC news article has been published to highlight the project's progress and the significant improvements made to GeoChemFoam. This media coverage has increased visibility among potential users and stakeholders in the HPC and simulation communities.

ARCHER2 Case Study:

We are in the process of preparing an ARCHER2 case study based on this project. This case study will document the deployment and performance of the GCF 5.1 and OpenFOAM 64-bit modules, providing practical insights and serving as a resource for future high-performance computing initiatives.

These outputs not only reflect the technical and scientific achievements of the project but also demonstrate our commitment to broad dissemination and community engagement, paving the way for future advancements in computational simulation on exascale systems.

Conclusion

This work has enabled vastly larger meshes as the memory of multiple nodes will become instantly available, enabling simulations on larger datasets, leading to more sophisticated research not just for GeoChemFoam users but for all the OpenFOAM community, especially those on the road to Exascale.

Preprint

https://arxiv.org/abs/2512.08438

Acknowledgement

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