Scalable and robust Firedrake deployment on ARCHER2 and beyond

ARCHER2-eCSE04-5

Jack Betteridge, Patrick E. Farrell, David A. Ham

30/4/2022

Abstract

We summarise the different aspects of Firedrake deployment that we have improved for our HPC users and additional benefits for ARCHER2 users.

1. A Spack package has been created for Firedrake many of its dependencies as well as determining a suitable Spack configuration and package workflow for ARCHER2.
2. Firedrake has been containerised for HPC resulting in a Singularity container suitable for use on ARCHER2.
3. A fix has been provided to PETSc/petsc4py to reduce the occurrence of deadlock issues when running Firedrake scripts in parallel on HPC systems.

1 Introduction

Firedrake is an automated system for the solution of partial differential equations using the finite element method (FEM) and sophisticated code generation. The goal for this project was to make installing and running the Firedrake framework[3, 13] — and its many dependencies — simple and robust on any HPC platform, with a primary focus on ARCHER2. We are confident in stating that this has been accomplished. Whilst there is still maintenance and upkeep of the devised solutions we have largely achieved what we set out to accomplish in this eCSE.
Namely we have:

1. Developed a Spack package for Firedrake.
2. Built a Singularity container for ARCHER2.
3. Improved the robustness of PETSc's Python support.

We have reordered these accomplishments in what follows to highlight the revised importance compared to the initial proposal. Namely that the development of a Spack package for Firedrake was a more complicated undertaking than originally thought, but additionally this work may have a larger impact for a wider range of ARCHER2 users.

2 Firedrake Spack Package

Spack[6, 12] is a popular choice of package manager for HPC users to install packages with complex dependencies. Firedrake previously did not support installation via Sapck, but it is one of the few package managers capable of delivering the fine grained control over build dependencies and is designed with HPC in mind. Furthermore, Spack can take advantage of any dependencies already available on a given system, either available through the OS or through the module system.

2.1 Previous situation

Prior to this work the installation path was the same on a HPC system as it would be on any other computer, namely use the Firedrake install script (listing 1). The `firedrake-install` command is a custom written

```
curl -O
    https://raw.githubusercontent.com/firedrakeproject/firedrake/master/scripts/firedrake-install
python3 firedrake-install
```

Listing 1: Firedrake install script install commands

Python script which can take numerous configuration arguments and is suitable for building Firedrake to the exact user specification.

Command line arguments can be used to specify the MPI distribution, an existing PETSc build if the user didn't want Firedrake to build its own, which BLAS/LAPACK libraries to link against and any other additional packages the user may want to install. Much of the work done by this install script is to overcome limitations in each dependency's own build system. It is important for Firedrake that everything is built using the same MPI distribution, since if the wrong distribution gets initialised by a package, PETSc will not be able to start. For instance, if `mpi4py` is built against OpenMPI and PETSc uses MPICH, the installation is broken. This is quite a common situation for other projects, but for Firedrake the situation is further complicated by dependencies in Python packages. For instance, PETSc and Numpy must both be linked against the same BLAS/LAPACK libraries, to prevent a FORTRAN ABI mismatch. If PETSc builds against a system install of NETLIB BLAS/LAPACK and Numpy uses the OpenBLAS bundled inside a pre-built wheel, again the installation is broken.

Many special configuration cases have been coded into `firedrake-install` to ensure that, on systems that we are aware of users targeting, the script generates a working Firedrake environment. The issue on HPC is when installation fails, users may be forced to rewrite parts on the installation script for the system they are on in order to get installation to succeed. For the end user the completed installation on their system is a Python virtual environment, which is self–contained as much as possible. This is the desktop user experience that we aim to recreate with the Spack package manager for HPC users.

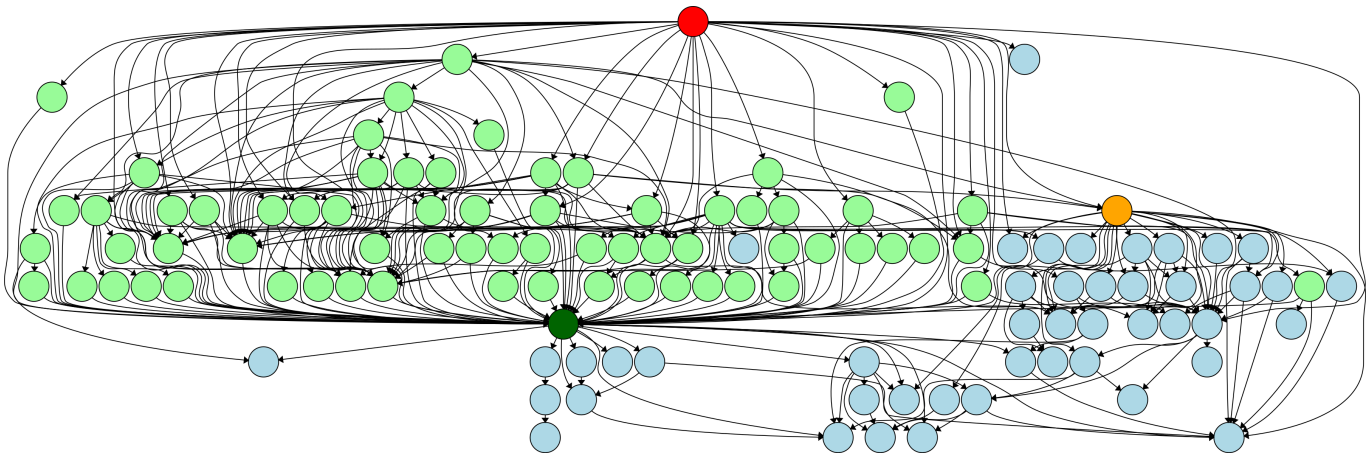

Figure 1: A directed graph showing the complexity of Firedrake's dependencies. Firedrake is highlighted at the top in red, Python as dark green and Python packages as light green. The orange circle is PETSc and all remaining pale blue circles are compiled dependencies. Generated by the command:
`spack graph -d py-firedrake %gcc ^mpich ^openblas > py-firedrake.dot`

2.2 Aims

From the outset we wanted any alternative installation to satisfy the following criteria:

1. To produce a working Firedrake installation on HPC machines.
2. To have the same (or as similar as possible) functionality to a regular script based Firedrake installation.
3. The new installation should require as little manual intervention as possible and should not require editing installation scripts or packages to succeed.

4. Fit entirely into the chosen (Spack) framework. That is, not to require additional functionality that isn't present in the existing framework.
5. To make the installation more extensible and better compatible with external and alternative package providers, as are often present on HPC.

2.3 Spack

Spack was chosen as the build framework due to its explicit support for HPC and ease of installation for a HPC user, rather than administrator. Spack already supports building many of Firedrake's dependencies, reducing the burden of maintenance for Firedrake developers. There is also a concerted effort in the long term maintenance of Spack, many of the US national laboratories are now using Spack and software sustainability groups such as xSDK[9, 10] require packages to be Spack installable. However, one criticism we have of the Spack package manager is the amount of work for a new user to learn and configure the tool. We outline configuration here and have detailed instructions available online[5, 4].

Before building Firedrake, Spack needs to be installed and configured to run on on ARCHER2. Spack can be installed on any UNIX or UNIX like system, but special care must be taken on a HPC facility to ensure functionality with the rest of the system packages.

The "installation" of Spack is straightforward, it requires cloning the Spack git repository and calling an activation script. Importantly, if there is no system installation of Spack, these steps can be performed easily as a HPC user without the need for administrative privileges. Spack even supports "chaining" installations, where one instance of Spack can use the packages from another instance. This will initially be very useful to us, as package updates that we have contributed are not yet in a versioned Spack release and many HPC systems will not update system modules unless there is a pressing reason to do so.

It is essential that a working Python interpreter is loaded as Spack is a Python program, the OS Python is often not sufficient — out of date and missing key internal components can prevent Spack from working correctly. It is our recommendation to load a Python module (such as `cray-python` on ARCHER2) if it is available before calling the spack activation script. After the activation step any Spack command can be invoked at the command line and users can start installing packages.

However, it is valuable to spend time configuring Spack for the HPC being used system.

2.4 Setup

Without any external configuration Spack uses whichever C, C++ and FORTRAN compilers are available on the path and will not use any system packages, instead it will bootstrap its own build system. While useful for systems with no package manager, or minimalist OS's, this is not ideal for HPC. On a supercomputer we wish to make use of specific optimised compilers, MPI distributions that take advantage of the interconnect hardware and libraries optimised for the architecture. Spack allows users to utilise this software, usually available as modules, through extensive configuration options.

The file[1] `$SPACK_USER_CONFIG/<system>/compilers.yaml` is read by Spack to determine what additional compilers are available on the system. On ARCHER2 we initially populated this file by running the command:

```
spack compiler find
```

Spack does a reasonable job of populating this file with the correct compiler specs as well as appropriate compiler flags, and even configures the correct modules to load to use the compilers. It is still worth checking this file carefully, as any error here may require all packages built with that compiler to be rebuilt.

A similar trick can be used to populate `$SPACK_USER_CONFIG/packages.yaml`, which is read by spack to determine additional packages that can be used. However,

```
spack external find
```

[1] `$SPACK_USER_CONFIG` if not explicitly set, will default to `~/.spack`. This may cause issues if you attempt to build packages on a computer node of ARCHER2 as the default home directory is not mounted.

only picks up build dependencies (tools like Bison, m4, git and tar), not the modules available on the system. Care must be taken to add `cray-mpich` as the MPI provider along with a suitable spec line, additional modules to load, the prefix path, as well as a tag to mark the package as not buildable by Spack. The same procedure must be done for the BLAS and LAPACK provider (`cray-libsci`) and the Python module, if they are desired as build dependencies, otherwise Spack will build its own copy.

Setting this up may be daunting for a user, but is possible to have a centrally installed instance of Spack or a centrally managed global Spack configuration (one that is overridden by the user's configuration, if desired). Once build configurations have been finalised the Spack settings documented on the Firedrake wiki[4] could be used as the ARCHER2 global configuration. Up until this point nothing has been Firedrake specific and is a useful introduction for anybody starting out using Spack on ARCHER2.

The Firedrake spack repository currently holds all of the additional packages currently required to build Firedrake. This repo works in conjunction with the Spack's `builtin` repo, offering additional packages and modifications of existing packages. Section 2.5 contains some examples.

Users can add the Firedrake Spack repo by cloning the remote repository and adding the repo to Spack's configuration:

```
git clone https://github.com/firedrakeproject/firedrake-spack.git
spack repo add firedrake-spack
```

In order to isolate the packages for the Firedrake installation in the same way that Python's venv does, the Spack installation uses Spack environments. This gives a very similar end user experience for anyone who has previously installed Firedrake using the install script. It also allows for packages to be added in the environment, with a prefix under the environment directory, giving a developer the same freedom to make changes to packages, whilst keeping all the core dependencies in one place.

The environment is created, activated and populated with the core packages by executing:

```
spack env create -d ./firedrake
spack env activate -p ./firedrake

spack develop py-firedrake@develop
spack develop py-pyop2@develop
...
```

Currently the core packages are added one at a time, and a typical Spack Firedrake environment consits of:

```
$ ls firedrake/
chaco        petsc   py-codepy py-fiat  py-firedrake py-islpy py-petsc4py  py-pyop2 py-ufl
libsupermesh py-cgen py-coffee py-finat py-genpy     py-loopy py-pyadjoint py-tsfc
```

At this point both the Spack build system and the Firedrake environment are configured and Firedrake can be installed.

2.5 Upstream changes

PETSc is one of the key dependencies of Firedrake and it is essential for core functionality that PETSc builds correctly, with all of its own dependencies and links against all libraries necessary for Firedrake to function. For stability, Firedrake maintains its own fork of PETSc tracking a few commits behind PETSc main, so it is necessary to create a modified package maintained in the Firedrake Spack repo that points to this fork. Changing the location of the source code could never be incorporated into the builtin Spack PETSc package, which necessitated its duplication in our own repository. A key contribution from this eCSE was the modification of the builtin PETSc package, to allow for inheritance and modification enabling the package to be subclassed without having to re-implement the install logic. This now allows other projects to create alternative PETSc installs using the Spack builtin as a parent class. Chaco, Eigen, NetCDF and

parallel NetCDF have been included as options in the Firedrake PETSc Spack package, since these libraries need to be linked against for a working installation.

Chaco, a mesh partitioner developed at Sandia, is an upstream package that has not seen active development for many years and did not previously have a Spack package[2], so this is also included in the Firedrake package repo.

In section 2.1 it was mentioned that Numpy (and Scipy) need to be linked against the same BLAS and LAPACK providers as PETSc to allow both to be imported in Python. This eCSE expanded the number of different BLAS/LAPACK implementations that Numpy could build against as options in Spack. This enables compatibility with both the AMD Optimizing CPU Libraries and Cray scientific libraries available on ARCHER2. Additional logic has also been added to the Scipy package to ensure that it too is built with the same BLAS/LAPACK as Numpy. These changes have been merged into the development branch of Spack for the benefit of all Spack users.

To allow for the use of different compilers in Spack, modifications had to be made to the PyOP2 package (which handles the compilation stage of the code generation within Firedrake). Refactoring allows end users to customise the compiler and compiler flags used with PyOP2 by creating new Python compiler classes, as well as allowing the default compiler classes to be overridden by environment variables. These environment variables can be automatically set when Spack loads the Firedrake environment. Now by default PyOP2 will use the same compiler that Spack used to build the rest of the Firedrake dependency tree.

Python packages within a Firedrake environment are installed in "developer mode", which allows for the source code in the cloned git repository to be used as if it were installed to Python's site-packages directory. This behaviour is not natively supported by Spack, but we have developed an `EditablePythonPackage` class in the `editable_install` package, which allows core dependencies to be installed in developer mode.

In addition to these changes the following is list of new or modified Spack packages that were created for this work and currently reside in the Firedrake Spack repository:

```
$ ls firedrake-spack/packages
chaco              py-cgen      py-firedrake      py-icepack   py-petsc4py   py-pytools
editable_install   py-codepy    py-folium         py-irksome   py-pulp       py-thetis
libspatialindex    py-coffee    py-genpy          py-islpy     py-pyadjoint  py-tsfc
libsupermesh       py-femlium   py-geojson        py-loopy     py-pygmsh     py-ufl
petsc              py-fiat      py-gmsh-interop   py-meshio    py-pymbolic   py-uptide
py-branca          py-finat     py-gusto          py-meshpy    py-pyop2      py-vtk
```

These will be offered to the maintainers of each project for their inclusion into Spack's builtin repository to ensure better package maintenance.

2.6 The `py-firedrake` package

The Spack package for Firedrake supports all the same functionality as the previous Firedrake install script including installing additional packages, specifying the MPI to use for dependencies and allowing foll customisation of PETSc build options. All this functionality fits into the Spack spec "language" for specifying versions and options for packages. For instance, if running on a local machine where the user wants Spack to build all dependencies, Firedrake can be configured using the GCC compiler, MPICH and OpenBLAS using the command:

```
spack add py-firedrake@develop %gcc ^mpich ^openblas
spack install
```

On ARCHER2, Firedrake can be configured with GCC, Cray Python, the Cray Scientific Libraries and Cray MPICH (if these have been added to the list of external packages) using the command in listing 2. Furthermore, the Firedrake Spack package includes additional functionality that cannot be added to the script. One additional feature is using system packages — and more importantly system modules — as part of a Firedrake build. Another is the ability to specify the compiler for the whole toolchain. When

[2] At least it didn't when this project started

```
spack add py-firedrake@develop \
    %gcc@10.2.0 \
    ^python@3.9.4.1 \
    ^cray-mpich@8.1.9%gcc@10.2.0 \
    ^cray-libsci@21.04.1.1
spack install
```

Listing 2: Spack configuration for ARCHER2 using GCC, Cray Python, Cray MPICH and Cray scientific libraries

the Firedrake installation script is used packages are not all guaranteed to use the same compiler. Changes to the PyOP2 package, as noted in section 2.5, make it possible for Spack to set the default compiler for PyOP2 to use for code generation when the Spack environment is activated.

Extensive instructions are currently held in a working document (currently available on hackmd), which will be added to the Firedrake wiki[4] and we hope to contribute these instructions and instructions for using Spack to the ARCHER2 documentation[2] site.

To test the portability of the Spack installer, it has also been successfully tested on several platforms. These include:

- Tier 2 HPC facility Isambard (XCI ThunderX2)
- HPC facilities at Imperial College
- HPC facilities at UCL
- Numerous end users personal machines, comprising different architectures

An additional configuration is being developed that utilises the NVidia compilers, to maximise performance on ARM hardware and will no doubt be useful in the future work porting Firedrake to GPUs as part of code generation's road to exascale.

We would like to acknowledge the contribution of Connor Ward, a PhD student who helped with the initial generation of many of the Spack packages used as part of this work.

3 Singularity

In order to support containerisation on ARCHER2, we need to create a Singularity image. For several years the Firedrake developers have been building several working Docker containers with many different flavours of Firedrake installation. These include:

- `firedrake-env` - The build environment, Ubuntu 20.04 with all required system packages, but without Firedrake.
- `firedrake-vanilla` - Firedrake with no additional packages or associated applications, which forms the base image for the remaining flavours.
- `firedrake` - Firedrake with some additional Firedrake applications bundled (Gusto, Thetis, Icepack, etc...).
- `firedrake-complex` - The same as `firedrake`, but with support for complex numbers.
- `firedrake-notebooks` - A container with both Firedrake and Jupyter notebooks installed, for running tutorials.

By using CI automation these Docker images can be kept up to date with every successful build.

Unfortunately, Docker is not a suitable container format for running on HPC as it allows for privilege escalation. The Apptainer/Singularity[3] project aims to perform the same role as Docker, but without the same security vulnerability so that images may be used on any HPC facility that supports containers.

In order to maintain a single source for the Firedrake container our work has focused on modifying the Docker container so that it may be converted to a usable Singularity container. Singularity allows images to be converted using the command in listing 3 (using the `firedrake-vanilla` container as an example

[3]Singularity used to be the name of both a closed source and open source project. These projects have diverged and the open source project is now called Apptainer and the closed source project is called Singularity. The executable is still called `singularity` in both cases, so we use this name in the report.

```
singularity pull firedrake-vanilla.sif docker://firedrakeproject/firedrake-vanilla
```

Listing 3: One step (pull and) conversion from Docker image format to Singulrity image format

throughout). However, this does not work for Firedrake images, since the `firedrake` virtual environment is built in the `$HOME` directory.

We have found that it is possible to convert the Docker image to an Singularity image in a two step procedure using a sandbox environment to prevent changes to the `$HOME` environment within the container:

```
1 singularity build --sandbox ./firedrake-vanilla
     docker://firedrakeproject/firedrake-vanilla
2 singularity build firedrake-vanilla.sif ./firedrake-vanilla
```

Listing 4: Two step procedure 1. (pull and) build with sandbox from Dockerhub, 2. Build Singulrity image format

Both of these commands can be executed without administrative privileges, so are still suitable for end users on ARCHER2. This saves both having to host the Singularity image or the user having to upload the large image from a personal machine.

The Docker images have been updated to include some additional system packages, such as the OpenFabric library, to enable the resultant Singularity image to be compatible with ARCHER2. Any of the Firedrake Docker images listed above can be converted using listing 4, not just `firedrake-vanilla`.

When configuring the image for running under MPI, there is a choice of hybrid or bind model. In the bind model the container has no MPI installation and instead the host system MPI is mounted within the container (AKA binding) and all packages in the image are linked against this MPI. Such a model is not suitable for Firedrake as it would require too many incompatible changes to the Docker images used elsewhere. Building a Singularity image from scratch is also not an option, since this process requires ARCHER2 to use the Cray's MPICH and would also need to be performed with administrative permissions (which we cannot have).

Instead we use the hybrid model, as recommended in the ARCHER2 container course, which involves using the MPI executable from the host. The application in the container is linked against and uses the MPI installation within the container which has the ability to communicate with the MPI daemon process running on the host system.

With the image built we can begin running applications in the Singularity container. Following ARCHER2 documentation[2, 1], we use `cray-mpich-abi` in place of `cray-mpich` and set the environment variables `SINGULARITYENV_LD_LIBRARY_PATH` and `SINGULARITY_BIND` appropriately. This enables additional Cray libraries and hardware specific device libraries to be found by executables inside the container.

Additional environment variables can be set within the container be prepending them with `SINGULARITYENV_`. We use this to control the location of the various cache directories, to ensure they are writable, and to locate the correct compilers inside the container:

```
1 export SINGULARITYENV_OMP_NUM_THREADS=1
2 export SINGULARITYENV_PYOP2_CACHE_DIR=/tmp/$USER/pyop2
3 export SINGULARITYENV_PYOP2_CC=/home/firedrake/firedrake/bin/mpicc
4 export SINGULARITYENV_PYOP2_CXX=/home/firedrake/firedrake/bin/mpicxx
5 export SINGULARITYENV_FIREDRAKE_TSFC_KERNEL_CACHE_DIR=/tmp/$USER/tsfc
```

Finally, when running Singularity additional steps must be taken to ensure that the Firedrake script is visible inside the container. By default Singularity will bind (mount within the container) various directories, including the host `$HOME` directory, and execute programs as the host system user (not the container user) using the mounted `$HOME` directory inside the container as the working directory. Care must therefore be taken to additionally bind any directories where scripts reside and where results will be saved, since an ARCHER2 user's `$HOME` directory is not mounted on compute nodes.

In the following example the `--home` argument binds the current working directory (`$PWD`) as the `$HOME` directory for the container, overriding the default behaviour. Additionally, the current working directory is

bound to the mount point `/home/firedrake/work`, using `--bind` so that scripts can be found when running in the container and results can be saved. The choice of these mount locations can be changed by the user to the location of their scripts, results directories and possibly a "home space" located under the `/work` directory as needed. Finally, the Python installed in the `firedrake` user's home directory is used to execute a script in the bound directory `/home/firedrake/work/`.

```
srun --ntasks-per-node 128  \
    singularity run --bind $PWD:/home/firedrake/work --home $PWD firedrake-vanilla.sif \
        /home/firedrake/firedrake/bin/python \
            /home/firedrake/work/myScript.py
```

Whilst there are a lot of options here, it gives the end user complete control over what locations on the ARCHER2 system are available for reading and writing by the container. Furthermore, using a container involves no installation for the user, just configuration.

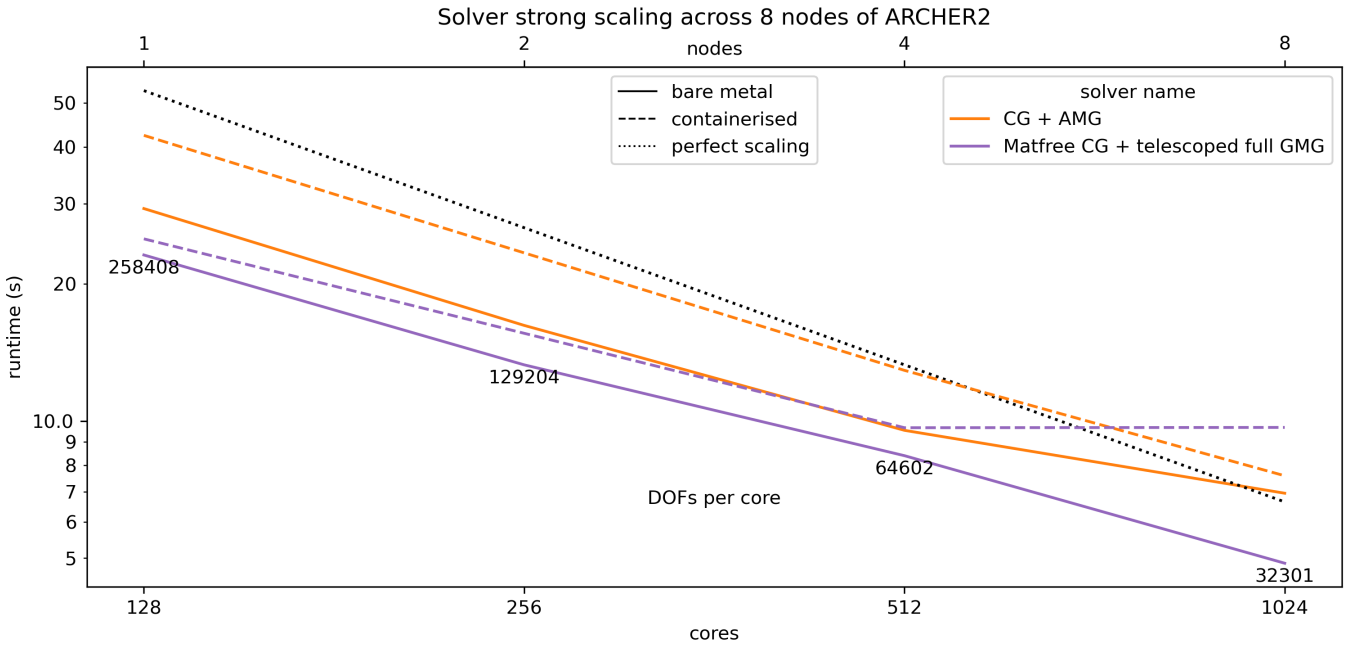

Figure 2: Strong scaling plot with a comparison of bare metal simulations (solid lines) and containerised simulations (dashed lines) for two different solvers. The CG solver with AMG incurs a 30–40% overhead running in a container on all but the 8 node simulation, the matrix free solver with GMG performs better with an 8–15% overhead on 1–4 nodes.

During multinode performance testing the Singularity container takes longer to execute simulations than when the same simulations are run on bare metal, as would be expected. From the experiments certain solvers perform better than others inside the container. Direct solvers seem to perform well and are close to bare metal performance, but are not scalable so cannot be used for large problems. However, scalable solvers of interest like multigrid seem to fare worse, although not terribly, as figure 2 illustrates. We postulate that this is due to way in which hierarchical solvers (like multigrid) are using MPI communicators within PETSc and it is the overhead of using certain operations in the containerised MPI installation that cause the slowdown.

Figure 2 shows a comparison between two of the solvers used for performance testing on 1–8 nodes of ARCHER2. The details of the solvers are not important for this discussion, but the conjugate gradient solver with algebraic multigrid preconditioning (CG + AMG) runs with a 30-40% overhead inside the container and a matrix free CG solver with telescoping geometric multigrid preconditioning (Matfree CG + telescoped GMG) runs with only an 8-15% overhead. On 8 nodes this situation is reversed, but this may be due to the tuning of the solvers for the specific problem size (notice that there are only 32000 degrees of freedom per core on 8 nodes in this experiment). For these tests we believe that this is an acceptable overhead, but

we aim to improve it in the future. It may unfortunately be the case that solvers require slightly different tuning for a containerised build compared to a bare metal installation.

It is clear from our investigation that internode communication is the biggest issue for the performance of solvers within containers, single node tests show very little overhead from using the Singularity container. We have only tried building a Singularity with the OpenFabrics network layer and we may yet see a difference with UCX network layer, although the configuration of such a container is undocumented. Further investigations into the best configuration is required, but rebuilding containers with a PETSc built with different options inside is incredibly time consuming. For now Singularity containers provide a zero installation route to using Firedrake, even if there is a small penalty to simulation performance.

4 Clean up of parallel objects in PETSc

Prior to the start of this eCSE we observed an issue when running Firedrake in parallel. When running on ARCHER2 (and other HPC facilities) jobs would randomly hang, with no apparent cause. On repeating the same job, it may then complete with no hang. Fixing the issue involved adding additional functionality to PETSc, which is tightly coupled to Firedrake and provides sophisticated, programmable solvers[8].

The source of this issue was determined to be Python's garbage collector. Specifically when running under MPI, Python objects created using petsc4py are allowed to be cleaned up by the cyclic garbage collector, causing simulations to deadlock. This happens because petsc4py objects are distributed across ranks and as such require synchronised destructon, but Python is not MPI aware and calls the garbage collector at different times on different ranks.

Whilst ARCHER2 isn't the first platform on which this problem has been observed, the combination of high core count per job and large number of Python objects created in certain algorithms meant a high prevalence of the issue during our previous simulations. The problem is very relevant to this eCSE as it affects both the scalability and robustness of Firedrake. Since the issue only occurs in parallel, and the chance that one Python instance calls the garbage collector out of turn increases with the number of MPI ranks used, any HPC platform is vulnerable to this issue. Furthermore, it can result extreme costs for HPC users as deadlocks do not terminate jobscripts. Instead it will waste all the Compute Units (CUs) assigned to a particular job and waste a lot of an end users time trying to track down the cause of the deadlock.

The solution we present performs the job of the cyclic garbage collector safely for Single Program Multiple Data (SPMD) programs. This is done in three stages:

1. **Creation** (Synchronous) At object instantiation the current number of objects associated with the current communicator is attained and incremented. This creation index is associated with that object for its lifetime.

2. **Delayed destruction** (Non-synchronous) At object destruction a reference to the underlying struct is held and added to a garbage hashmap using the creation index as the key. The memory is not yet freed as a reference to the struct persists. Since this step is non-synchronous it can safely be called at any time by Python's garbage collector.

3. **Cleanup** (Synchronous) A cleanup routine is periodically called on all ranks within the communicator to deallocate the memory occupied by the structs present in the garbage hashmap that *have been destroyed on all ranks* and *in ascending creation index order*. Algorithm 1 outlines the procedure used for this step.

These changes are completely transparent to the end user, when this fix gets merged into PETSc no changes will need to be made to existing codes. At this point Firedrake scripts will no longer hang in parallel due to the garbage collection issue, preventing wasted resources when running jobs. By merging this fix into upstream PETSc, it will benefit all petsc4py users, not just Firedrake users.

Preliminary results indicate that the fix will have little impact on the performance of code after the fix is merged. Figure 3 shows the time to create 100 distributed objects (bulk time) and the time taken to clean up the memory from the objects when they have been destroyed (cleanup time). These times are presented for two different methods, the two plots on the left are indicative of the performance before the fix on experiments are prone to hang, the plots on the right show the performance with the fix implemented.

Algorithm 1: Parallel garbage collection function

```
cleanup(comm)
    garbage = getGarbage(comm)
    sorted_keys = sort(garbage.keys())
    intersection = gatherIntersect(comm, sorted_keys)
    for key in intersection do
        object = garbage.get(key)
        object.collectiveDestroy()
        garbage.remove(key)
    end
return
```

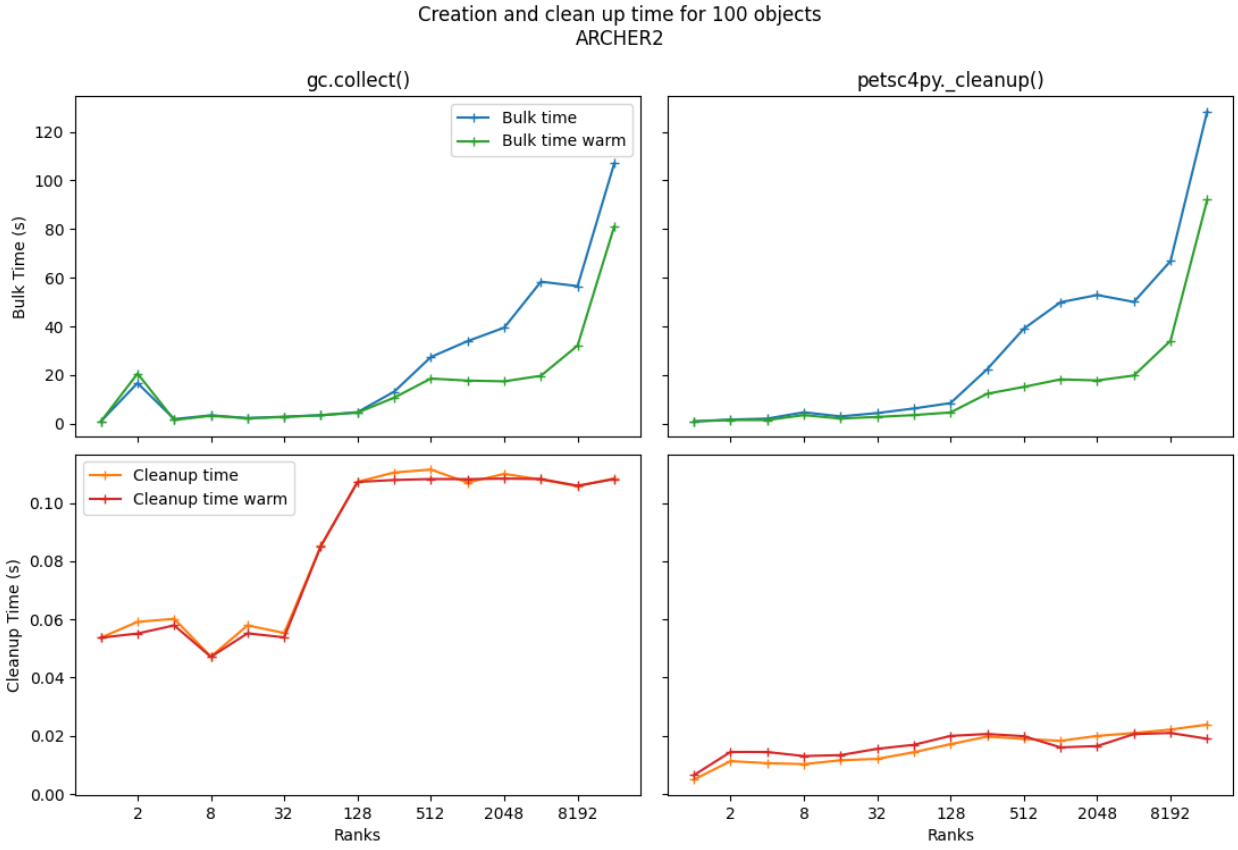

Figure 3: Performance comparison of bulk (object creation time), both top, and cleanup time, both bottom, for simulations using the broken object cleanup procedure, both left, compared to the newly implemented parallel safe cleanup, both right. Runs were performed with both cold and warm caches.

Two lines are shown on each plot, the time to create/destroy the objects with a cold cache (simply labelled 'time') and time to create/destroy the objects on a second run with populated Python caches (labelled 'time warm').

The techniques used in this improvement to PETSc/petsc4py will be explored in an upcoming publication.

A Appendix

A.1 Spindle

Originally we envisioned a launch utility that would handle process distribution (like `mpiexec`), process pinning (like `likwid-pin`) as well as improving dynamically linked library load time (like `spindle`).

Such a tool would prevent commands such as

```
spindle --x \
    likwid-pin -N:0-128\
        mpiexec -n 1024 -ppn 128 -bind-to \
            python -B -m memory_profiler\
                my_script.py -pc_mg_log -log_view :my_script.txt:ascii_flamegraph
```

We decided against creating a generic runscript as placing all the different options together would serve no purpose. Such a utility may also cause further issues due to providers for some tools MPI providers and functionality overlap between different tools (aprun and likwid for instance perform process placement differently).

Unfortunately, we were unable to get Spindle[11, 7] to work on ARCHER2. Installing via Spack did not produce a useable program, more system configuration was required. Manually installing threw up further issues. Firstly, Spindle is not compatible with the version of SLURM installed on ARCHER2. While the installation can continue and there are suggested workaround flags for SLURM, the workaround is ineffective and Spindle does not work correctly. Secondly, the configuration of Spindle seems to depend on certain communication features of the system that may be disabled for security on ARCHER2.

Spindle is an interesting package that we may investigate in the future. We cannot bundle it as a user installable component of Firedrake, but it may be of interest to the ARCHER2 CSE team as a tool to improve library loading. Such a tool would need to be provided as a system installation to be truly effective.

An ad hoc alternative to Spindle is outlined in appendix A.2.

A.2 Tarballing

One method that we can use to prevent large jobs thrashing the network filesystem is to create installations (either a Spack install or script based install) in a directory in `/tmp`. When the installation is complete the whole environment can be stored compressed as a single tarball file somewhere under the `/work` directory. This modification to the installation is shown in listing 5.

```
...
mkdir -p /tmp/$USER
spack env create -d /tmp/$USER/firedrake
spack env activate -p /tmp/$USER/firedrake
...
spack install --fail-fast 2>&1 | tee $SPACK_ENV/spack-firedrake-install.log
tar -czvf /work/PRJ/PRJ/$USER/firedrake.tar.gz /tmp/$USER/firedrake
```

Listing 5: Modified steps in Spack installation, showing installation under `/tmp` and tarballing the installation at the end (other steps elided)

It is advantageous to tarball the environment as this single file can be sent to all compute nodes at run time (once per node) and then each rank per node can load files from the local filesystem or ramdisk. This is in contrast to all ranks on all nodes trying to read from the network filesystem, which causes a bottleneck. On ARCHER2 each node has 128 physical cores, which have to share just 2 network cards. By

predistributing the environment the load on the network filesystem at initialisation time is reduced by a factor of 128.

```
source /work/PRJ/PRJ/$USER/spack/share/spack/setup-env.sh
srun --ntasks-per-node 1 tar -xzf /work/PRJ/PRJ/$USER/firedrake.tar.gz -C /tmp/$USER
spack env activate /tmp/$USER/firedrake-env/gcc
python myScript.py
```

Listing 6: Modified steps in running script under Spack, untarballing the installation before activating the environment

However, installing in this manner makes modifying the installation cumbersome as the tarball must be unpacked to `/tmp`, before any changes can be made to the installation. Once changes are made, it is also necessary to re-tarball the whole installation, overwriting the original tarball. This is not an ideal solution for developers or for end users due to the extra work involved to modify and activate the environment. We are actively seeking an alternative method to replace tarballing the whole environment.

References

[1] ARCHER2 container course material. https://epcced.github.io/2020-12-08-Containers-Online/. Accessed: 2022-05-31.

[2] ARCHER2 documentation. https://docs.archer2.ac.uk/. Accessed: 2022-05-31.

[3] Firedrake website. https://www.firedrakeproject.org/. Accessed: 2022-05-31.

[4] Firedrake wiki. https://github.com/firedrakeproject/firedrake/wiki. Accessed: 2022-05-31.

[5] HackMD working document for installing Firedrake with Spack on ARCHER2. https://hackmd.io/Sg3fYXuCTl61d_LAg4QnMw?both. Accessed: 2022-05-31.

[6] Spack website. https://spack.io/. Accessed: 2022-05-31.

[7] Spindle website. https://computing.llnl.gov/projects/spindle. Accessed: 2022-05-31.

[8] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. D. Gropp, D. Karpeyev, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes, R. T. Mills, T. Munson, K. Rupp, P. Sanan, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.15, Argonne National Laboratory, 2021.

[9] R. Bartlett, I. Demeshko, T. Gamblin, G. Hammond, M. Heroux, J. Johnson, A. Klinvex, X. Li, L. C. McInnes, J. D. Moulton, D. Osei-Kuffuor, J. Sarich, B. Smith, J. Willenbring, and U. M. Yang. xsdk foundations: Toward an extreme-scale scientific software development kit, 2017.

[10] D. E. Bernholdt, M. Heroux, D. S. Katz, A. Logg, and L. C. McInnes. Posters presented at SIAM CSE17 PP108 Minisymposterium: Software productivity and sustainability for CSE and data science, Feb 2017.

[11] W. Frings, D. H. Ahn, M. LeGendre, T. Gamblin, B. R. de Supinski, and F. Wolf. Massively parallel loading. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 389–398, 2013.

[12] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral. The Spack Package Manager: Bringing Order to HPC Software Chaos. Supercomputing 2015 (SC'15), Austin, Texas, USA, November 15-20 2015. LLNL-CONF-669890.

[13] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. McRae, G.-T. Bercea, G. R. Markall, and P. H. Kelly. Firedrake: automating the finite element method by composing abstractions. *ACM Transactions on Mathematical Software*, 43(3):1–27, 2016.