# An ARCHER2 User's Spack Primer

William Lucas, Luca Parisi
EPCC, The University of Edinburgh

w.lucas@epcc.ed.ac.uk, l.parisi@epcc.ed.ac.uk

archer2

epcc

# ARCHER2 Partners

epcc


epcc


THE UNIVERSITY *of* EDINBURGH

Hewlett Packard Enterprise

# Outline

- What is Spack?
- Spack basics
- Using Spack on ARCHER2
- Packages and repositories
- Environments and modules
- Live demonstration

# What is Spack?

# Package managers

- Examples: APT, DNF, Zypper, Portage, Flatpak, Fink, Homebrew
- How do they work?
  - You tell it you want to install a software package
  - It works out what dependencies are needed,
  - as well as what dependencies those need,
    - Directed Acyclic Graph (DAG)
  - then installs them all from the bottom up.
- Examples above target various OSs, distribution styles, provide binaries/compile from source/sandboxed installs
- All target 'normal' desktop and server use.
- What about HPC?

# The Spack package manager

epcc

For HPC, we'd typically like to:

- Have access to packages which provide the libraries and applications that are used in HPC.
- Compile those packages from source for performance.
- Be able to use important external libraries (MPI) as dependencies where required, again for performance.
- Simultaneously install packages at different versions with different options at top level and in dependencies.

Spack was designed to do this.

# Spack basics


epcc

# How to use Spack

- Spack commands can be as simple as:
  > spack install gromacs
  - Download GROMACS and dependencies' source code, compile them all using defaults as far as they can be made self-consistent, and install.
- Install location is user configurable.
  - A fresh Spack installs to a directory within itself
  - On ARCHER2 it is /work/<project>/<project>/<user>/.spack
- These locations are not added to the PATH by default.
- In practice, installed software is made available for use through load commands, either spack load or module load.

# Spack specs

- A spec is a package name (‘gromacs’) followed ***optionally*** by any of its version, build options (‘variants’), the name, version and build options of any dependencies, and the compiler to use and its version.

| Symbol | Meaning |
|---|---|
| @<version number> | Package or compiler version. |
| +<build variant> | Enable this build variant. |
| ~<build variant> | Disable this build variant. |
| <variant name>=<option> | Set non-Boolean variant option. |
| ^<dependency spec> | Use this dependency for the concretisation and build. |
| %<compiler spec> | Use this compiler for the concretisation and build. |

- *e.g.* gromacs@2023.3 +mpi ~openmp %gcc ^fftw@3.3.10
- Propagate variant options through dependencies with ++, ~~, --, ==

# Spack concretiser (aside)

- Dependencies form a directed acyclic graph (DAG).
- Given a spec, Spack must determine a full and self-consistent DAG of all packages to be installed and their versions and variants – this is called concretisation.
- Spack is Python; the concretiser is Answer Set Programming (ASP).
  - Facts: cat is 5kg, dog is 15 kg, horse is 500 kg:
    weight(cat,5). weight(dog,15). weight(horse,500).
  - Rule: a heavy animal A is one which has a weight W and for which W is > 100 kg:
    heavy(A) :- weight(A,W), W > 100.
  - The grounder replaces variables (here A and W) with facts.
  - The solver produces a set or sets of true facts to solve the program:
    weight(cat,5) weight(dog,15) weight(horse,500) heavy(horse)
  - Combined grounder and solver 'clingo' downloaded on first Spack run.

# Spack configuration

- Configured through sets of YAML files spanning several scopes.
- At the top is user-level scope: the user can override any options.
- Spack is configured on ARCHER2 to:
  - Use the system compilers (GCC 11.2.0, CCE 16.0.1, AOCL 4.0.0)
  - Use libraries from the Cray Programming Environment (MPICH, LibSci, FFTW, HDF5, NetCDF)
  - Make available several custom packages written for ARCHER2
  - Install packages in users' /work directories so they can be used in jobs

# Basic Spack workflow

- Find the package you want to install and then get information on versions available, variants, defaults, dependencies:
  > spack list zlib
  > spack info zlib-ng
  - or use the website https://packages.spack.io/
- See what *would* be installed given a spec (concretise it):
  > spack spec zlib-ng %gcc
- Actually install it:
  > spack install zlib-ng %gcc

# Useful commands

epcc

| Command | Use |
|---|---|
| spack help [command] | Get help in general or on given command. |
| spack list <package-name> | Find available packages containing this name. |
| spack info <package-name> | Show information on the given package. |
| spack spec <package-spec> | Concretise and display the given package spec. |
| spack install <package-spec> | Concretise and install the given package spec. |
| spack find [package-spec] | Show installed packages matching the optional spec or all installed if none. |
| spack config get <config-section> | Show given configuration (config, packages, compilers, repos...) |
| spack config blame <config-section> | Show which config files contribute to which lines of the active config. |
| spack repo create <directory> [namespace] | Create repository at the given location and with optional given namespace. |
| spack repo add <directory> | Add the given repository to the user scope's repos.yaml. |
| spack create <download URL> | Create a new package after scraping information from the download URL. |
| spack edit <package-name> | Open the package in text editor. |

# A note on distinguishing packages

If you only have one installation of a given package, you can use its simple spec to refer to it:

```
> spack find zlib-ng
-- linux-sles15-zen2 / %c,cxx=gcc@11.2.0 --------------
zlib-ng@2.2.4
==> 1 installed packages
> spack uninstall zlib-ng
```

# A note on distinguishing packages

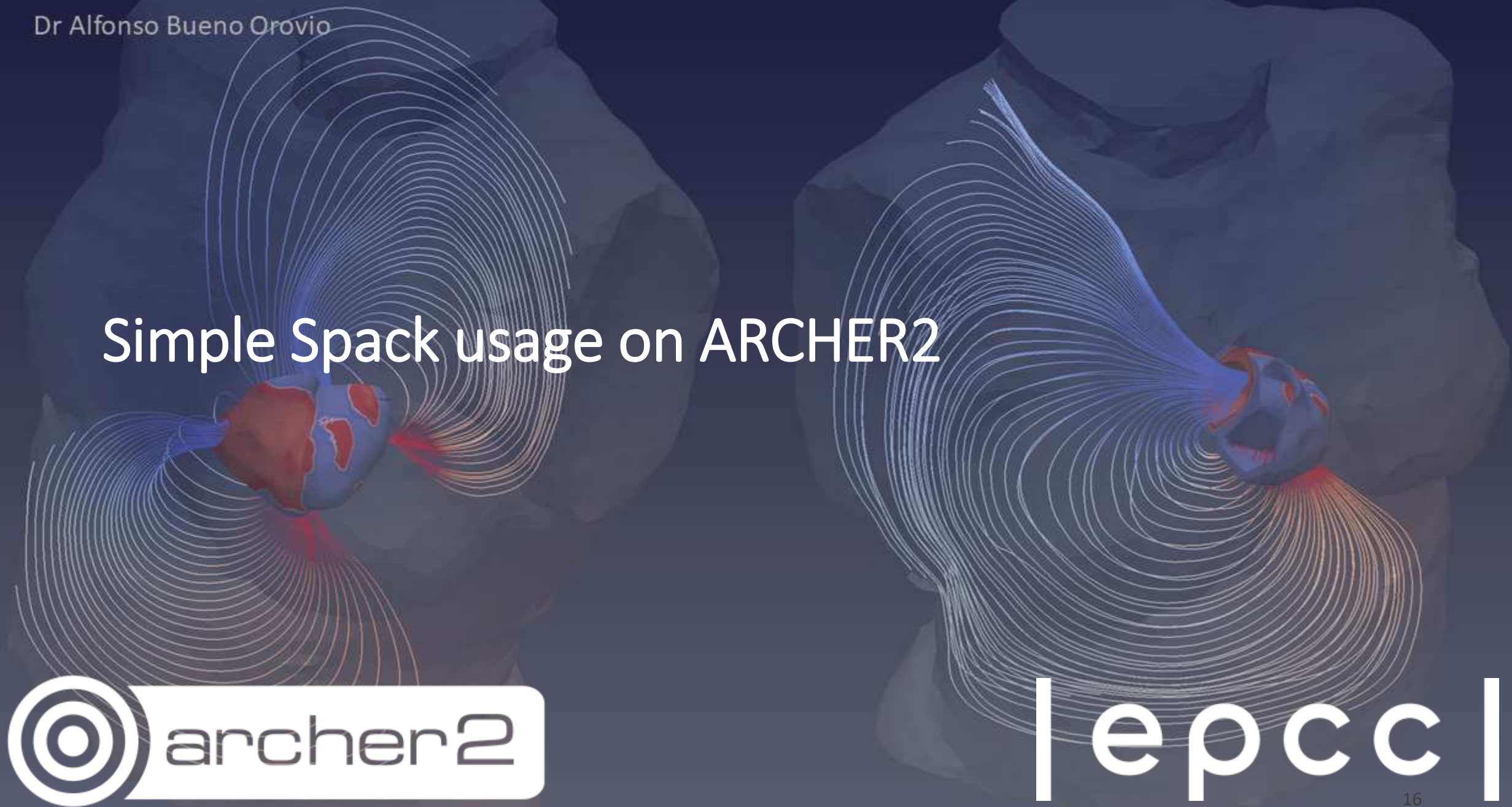
But if there are multiple installations of a package, you can distinguish them using their unique hashes. spack find –l gives these:

```
> spack find -l zlib-ng
-- linux-sles15-zen2 / %c,cxx=gcc@11.2.0 --------------
heo7tra zlib-ng@2.2.4  clsyri4 zlib-ng@2.2.4
==> 2 installed packages
> spack diff /heo7tra /clsyri4
[snip output showing what's different]
> spack uninstall /clsyri4
```

Prepend hash with a forward slash to use it in place of a spec

Dr Alfonso Bueno Orovio

# Simple Spack usage on ARCHER2



# Activating Spack on ARCHER2

Spack is available for users through a module:

1. Load other-software module:
   > module load other-software
   Warning: You have enabled access to software packages installed by external parties on ARCHER2 or those that are not fully-supported.
   ...
2. Load Spack module:
   > module load spack
3. Go ahead and use Spack to do something:
   > spack install quantum-espresso@7.4.1

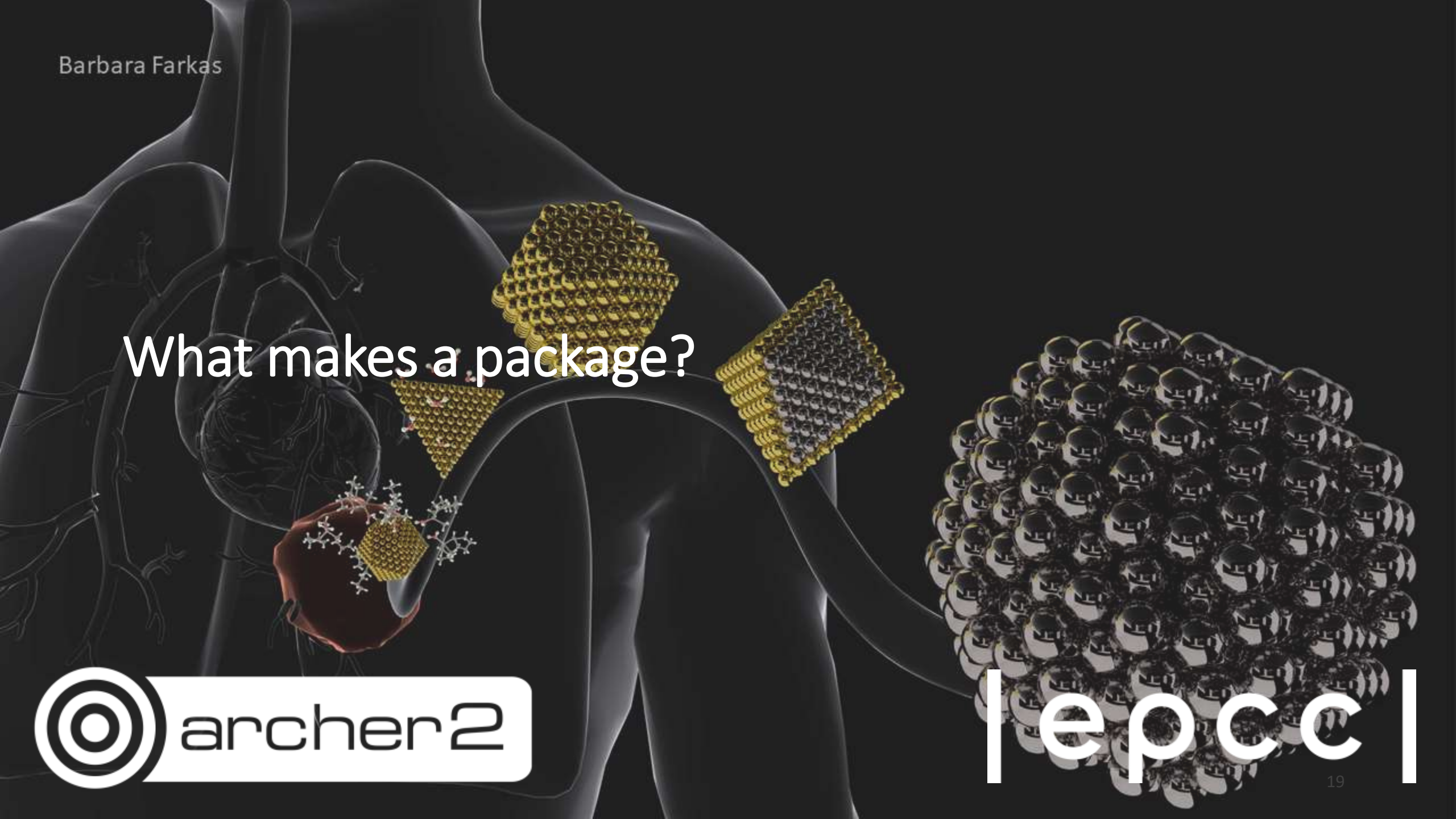
# Running in jobs

## Simple way to use Spack in jobs is like so:

```
#!/bin/bash

#SBATCH --job-name=qe_spack
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=128
#SBATCH --cpus-per-task=1
#SBATCH --time=00:20:00

#SBATCH --partition=standard
#SBATCH --qos=standard


# Load quantum_espresso via Spack

module load other-software
module load spack
spack load quantum_espresso

export OMP_NUM_THREADS=1
export SRUN_CPUS_PER_TASK=$SLURM_CPUS_PER_TASK

srun --hint=nomultithread --distribution=block:block pw.x < test_calc.in
```

Barbara Farkas

# What makes a package?



archer2

epcc

# Package structure

- Package are directories within a repository's packages directory.
- These directories are named after the software the package installs.
- Each package directory needs to contain a package.py Python script which tells Spack how to install the software.
  - Package information
  - Versions
  - Source code download URL
  - Build system and options – Autotools, CMake, SCons, Ninja...
- It may also contain patches to apply to source code.

# Creating a package from scratch

epcc

- You can write packages of your own for any software.
- Spack can take a lot of pain out of the start with spack create.
- Provide a download URL and Spack will find all the versions it can, and after checking what versions you want, download the source to calculate checksums to place in the package.py:
  spack create https://www.package.org/source-1.0.tar.gz
- If you'd rather just get the boilerplate in place and then start work yourself, provide the name instead of a source URL:
  spack create --name packagename
  - Then edit in default text editor with spack edit packagename.

# Spack package repositories

- ‘Repository’ in the dictionary sense rather than a version control repository: a location (local directory, maybe a URL) containing sets of package installation instructions.
- A repository has a namespace which Spack uses to refer to the packages it provides – often just the name of the directory it's in.
- An installed package has a full name which indicates the namespace of the repository which was used to install it, *e.g*. a package could be builtin.quantum_espresso if using the builtin repository, or archer2.quantum_espresso if using the archer2 repository.

# Creating a repository

- Create a repository with
  spack repo create /work/path/to/repository
  where the path can be relative or absolute.
- If you'd like to give it a namespace which isn't the directory name, it's just an extra argument to the command:
  spack repo create /work/path/to/repository namespace
- NB it is possible to create the repository manually, but the Spack command correctly creates all the internal structure so is preferred.

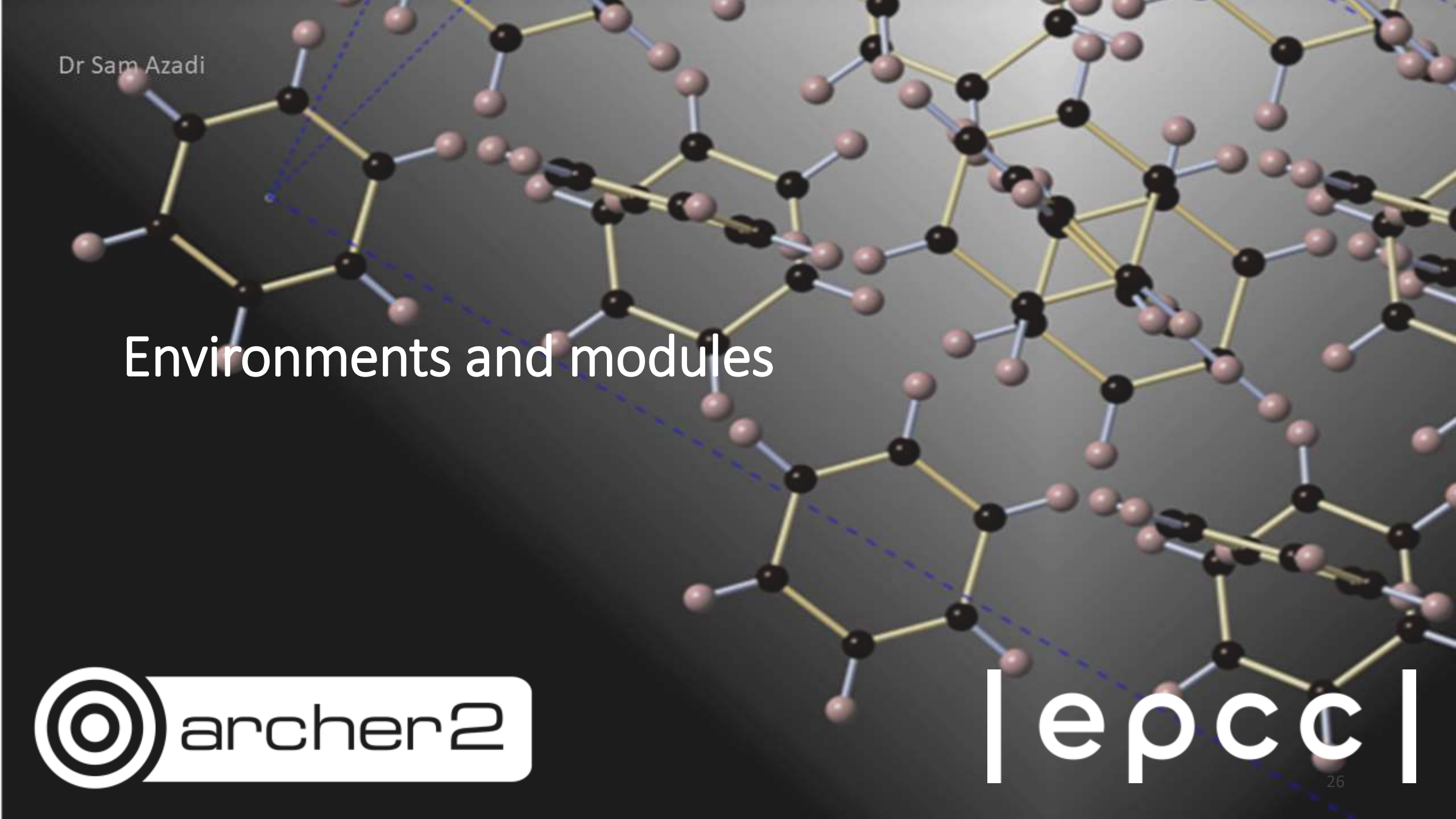
# Activating a repository

- When Spack is asked to concretise a spec, it will search whichever repositories it is configured to use in the order they are provided in the repos.yaml config to find a matching package.
  - Until the new repository is in repos.yaml, Spack won't use it.
- Add a repository to user space repos.yaml with:
  spack repo add /path/to/repository/directory
- And remove it again, if necessary, with:
  spack repo rm /path/to/repository/directory
- You can directly edit repos.yaml if you want to.

# The builtin repository

- The builtin repository contains 8834 packages as of 15 April.
- Searchable at https://packages.spack.io/.
- Used to be contained within the main Spack GitHub repository, but is now its own repository
  - The term 'repository' is slightly overloaded here...
- An example of how you can indeed provide a GitHub URL for a package repository in repos.yaml:

```
> spack config get repos
repos:
  archer2: ${SPACK_EPCC_REPOS}/archer2/spack_repo/archer2
  builtin:
    git: https://github.com/spack/spack-packages.git
    branch: releases/v2025.07
```



Dr Sam Azadi

# Environments and modules

archer2

epcc

# Spack environments

- It's good practice to create and install into an environment
  - Very similar conceptually to a Conda environment
  - A self-contained collection of software activated as a group
- On the file system, an environment is set up as nothing more than a directory containing a spack.yaml file, itself containing the list of package specs to be installed (and any other options).
  - By default within the Spack directory tree, but can be anywhere.
  - Once the environment is activated, software is installed in this directory.
- Use cases:
  - Manage coherent software environments (release, update, deprecate)
  - Share sets of packages amongst project users
  - Provide environment as software distribution method

# Environment usage

- Create environment with:
  - Environment in default directories: spack env create <env name>
  - Elsewhere: spack env create /work/path/to/directory
- Activate environment with:
  - Environment in default directories: spack env activate <env name>
  - Elsewhere: spack env activate /work/path/to/directory
- Modify spack.yaml directly or add package spec with
  > spack add <spec>
- Install everything in the environment:
  > spack install
- Deactivate environment:
  > spack env deactivate

# Simple environment

Can also open environment config in default editor (vim) with
> spack config edit

Example spack.yaml environment configuration:

```
spack:
  specs:
  - petsc %gcc@11.2.0
  - petsc %cce@15.0.0
  - nektar@5.5.0 +vortexwave_solver
  - nektar@5.4.0 +pulsewave_solver
  view: true
  concretizer:
    unify: when_possible
```

# Simple environment

Can also open environment config in default editor (vim) with
> spack config edit

Example spack.yaml environment configuration:

```
spack:
  specs:
  - petsc %gcc@11.2.0
  - petsc %cce@15.0.0
  - nektar@5.5.0 +vortexwave_solver
  - nektar@5.4.0 +pulsewave_solver
  view: true
  concretizer:
    unify: when_possible
```

Install these package specs in the environment when running spack install.

# Simple environment

Can also open environment config in default editor (vim) with
> spack config edit

Example spack.yaml environment configuration:

```
spack:
  specs:
  - petsc %gcc@11.2.0
  - petsc %cce@15.0.0
  - nektar@5.5.0 +vortexwave_solver
  - nektar@5.4.0 +pulsewave_solver
  view: true
  concretizer:
    unify: when_possible
```

Produce a traditional tree of bin, lib, share directories containing symbolic links to the actual files.

# Simple environment

Can also open environment config in default editor (vim) with
> spack config edit

Example spack.yaml environment configuration:

```
spack:
  specs:
  - petsc %gcc@11.2.0
  - petsc %cce@15.0.0
  - nektar@5.5.0 +vortexwave_solver
  - nektar@5.4.0 +pulsewave_solver
  view: true
  concretizer:
    unify: when_possible
```

If possible, reuse package dependencies across the specs.

# Module generation

Spack can automatically create modules. Write a modules.yaml configuration, or add as options inside an environment's spack.yaml:

```
spack:
  [snip...]
  modules:
    default:
      enable: [lmod]
      roots:
        lmod: /work/path/to/modulefiles
```

# Module generation

- Options like the previous will create modules at the point that you run the spack install command.
  - If you need to regenerate modules (e.g. you've changed the options):
    > spack module lmod refresh --delete-tree –y
- Other options available include:
  - Module naming convention.
  - Which modules should be treated as defaults by module commands.
  - Which environment variables should generally be set in modules.
  - Package-specific environment variables to be set.

# Module use

- Once the modules are in place, make them available with one of:

```
> module use /work/path/to/modulefiles
> export MODULEPATH=/work/path/to/modulefiles:$MODULEPATH
```

- If in a project shared directory with group read permissions, you can pass these to others in your group for them to also use.
- Then use just as you would any modules:
  - module avail …
  - module load …
  - module unload …
  - module swap …


Live demonstration
archer2
epcc