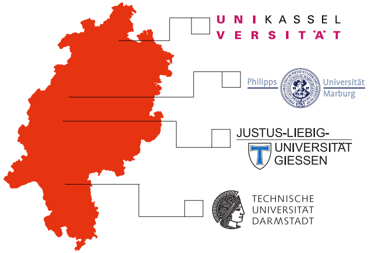
Workflows on HPC Systems

Hessisches Kompetenzzentrum für Hochleistungsrechnen (HKHLR)

T. Jammer R. Sitt Dr. C. Iwainsky


UNI KASSEL VERSITÄT
Philipps Universität Marburg
JUSTUS-LIEBIG-UNIVERSITÄT GIESSEN
TECHNISCHE UNIVERSITÄT DARMSTADT

HKHLR is funded by the Hessian Ministry of Sciences and Arts

HESSEN

Course Contents

Template for a scientific HPC workflow

- How to upload the data to the cluster in a structured way
- Pre-processing of the data
- Organisation of calculations
- Post-processing / aggregation
- Documentation and archiving of the workflow (following FAIR Principles)

- Focus on practical examples

Agenda

Morning session

- A scientific HPC workflow template
- Mapping a “problem” to the workflow: a practical example
- Introduction to the OpenFOAM example

DYI exercise session

- Exercise: map OpenFOAM example to the workflow

Afternoon session

- Mob-coding: solving the exercise
- Wrap-up / Q&A

Motivation & FAIR Principles

Scientific Method:

- Experiments can be repeated
- Results can be reproduced

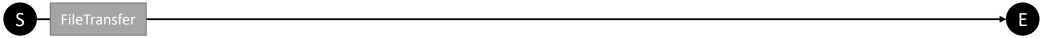
FAIR Principles

- **F**indable
- **A**ccessible
- **I**nteroperable
- **R**eusable

Terminology

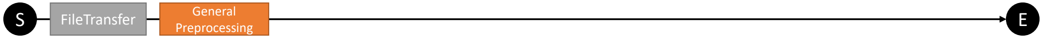
- Problem:
  - a research question, or particular computation need.
  - de-composed in 1 or more runs;
- Run:
  - complete sub-component of the problem, that can be computed front to back
  - possible daisy chains
  - split in separate aspects

The general workflow



▨ Transfer from and to the HPC system and the users device

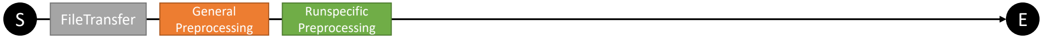
The general workflow



⬜ Transfer from and to the HPC system and the users device

🟧 Generic pre-&post-processing of data, like generating the mesh

The general workflow

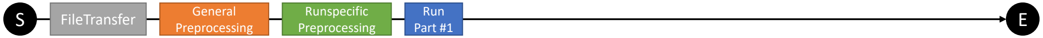


⬜ Transfer from and to the HPC system and the users device

🟩 Case/Run specific pre &- post processing

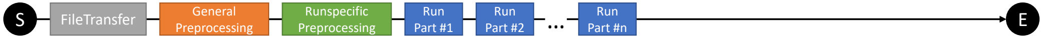
🟧 Generic pre-&post-processing of data, like generating the mesh

The general workflow



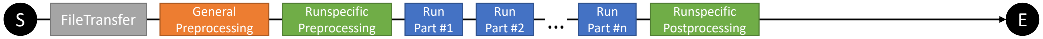
⬜	Transfer from and to the HPC system and the users device
🟧	Generic pre-&post-processing of data, like generating the mesh
🟩	Case/Run specific pre &- post processing
🟦	Possible concurrent, possibly segmented HPC work

The general workflow



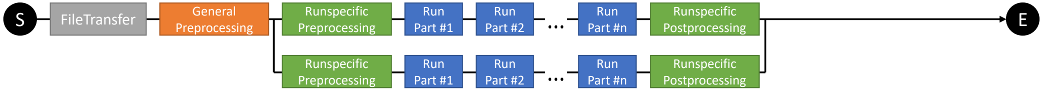
⬜	Transfer from and to the HPC system and the users device
🟧	Generic pre-&post-processing of data, like generating the mesh
🟩	Case/Run specific pre &- post processing
🟦	Possible concurrent, possibly segmented HPC work

The general workflow



⬜	Transfer from and to the HPC system and the users device
🟧	Generic pre-&post-processing of data, like generating the mesh
🟩	Case/Run specific pre &- post processing
🟦	Possible concurrent, possibly segmented HPC work

The general workflow



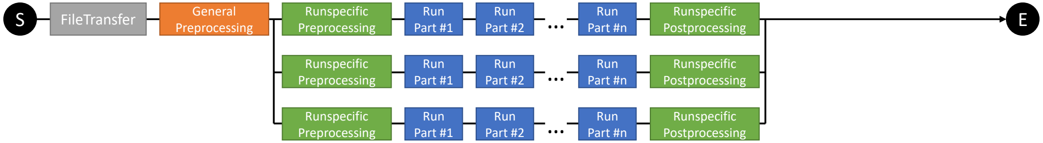
⬜ Transfer from and to the HPC system and the users device

🟧 Generic pre-&post-processing of data, like generating the mesh

🟩 Case/Run specific pre &- post processing

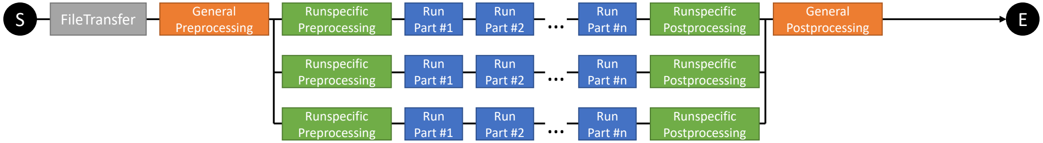
🟦 Possible concurrent, possibly segmented HPC work

The general workflow



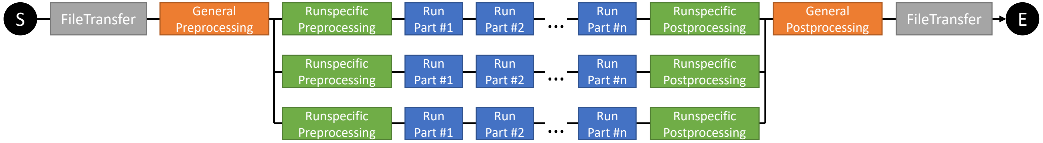
⬜	Transfer from and to the HPC system and the users device
🟧	Generic pre-&post-processing of data, like generating the mesh
🟩	Case/Run specific pre &- post processing
🟦	Possible concurrent, possibly segmented HPC work

The general workflow



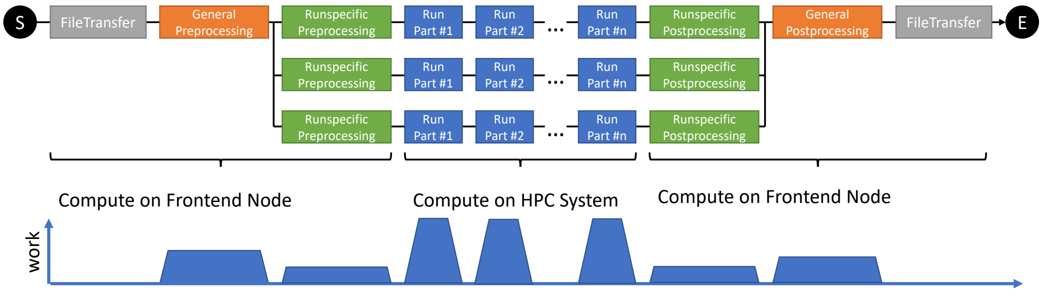
⬜	Transfer from and to the HPC system and the users device
🟧	Generic pre-&post-processing of data, like generating the mesh
🟩	Case/Run specific pre &- post processing
🟦	Possible concurrent, possibly segmented HPC work

The general workflow



- ⬜ Transfer from and to the HPC system and the users device
- 🟧 Generic pre-&post-processing of data, like generating the mesh
- 🟩 Case/Run specific pre &- post processing
- 🟦 Possible concurrent, possibly segmented HPC work

The general workflow



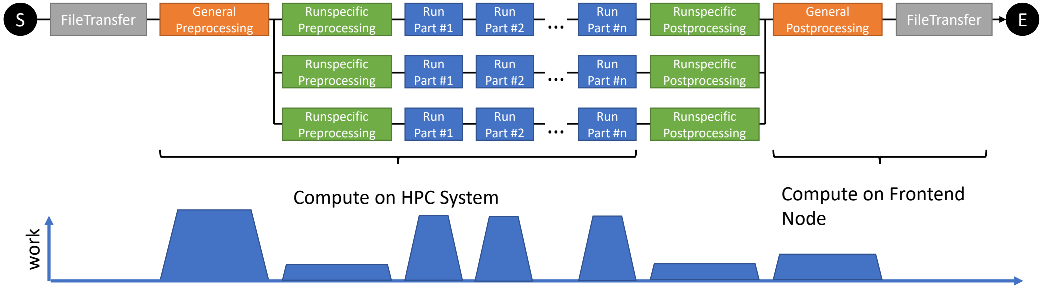
⬜ Transfer from and to the HPC system and the users device

🟧 Generic pre-&post-processing of data, like generating the mesh

🟩 Case/Run specific pre &- post processing

🟦 Possible concurrent, possibly segmented HPC work

The general workflow



⬜ Transfer from and to the HPC system and the users device

🟩 Case/Run specific pre &- post processing

🟧 Generic pre-&post-processing of data, like generating the mesh

🟦 Possible concurrent, possibly segmented HPC work

Ideal configuration

An ideal implementation of the workflow pipeline can utilize:

- multiple independent instances of computation
- segment a single computation into multiple subsequent iterative steps
- failure recovery to restart instances in case of failure
- incorporate new data-sets after principal computation has started
- documents whole research pipeline as a digital artifact
- generates final results directly from data without manual intervention

Example problem: machine-learning of Housenumbers

- General problem description:
- Train Deep Neural Network to regognize what number is displayed
- Input: Image of a house number sign, shield, emblem, etc.
- Output: The number displayed on this house
- Input parameter: Hyperparameter to train the model (e.g. the number of neurons)
- Task: Find the best hyperparameters for maximum model efficiency

Image from : http://ufldl.stanford.edu/housenumbers, Non commercial use only

Sidenote on software

Software sources:

Check local availability

- local “free-to-use” software
- local preinstalled commercial software with licenses
  - ‘module’-system
- self-installed software
  - packaged software PIP, spack, easybuild
  - self-compiled software
  - binary releases (TAR, o.s.)

Scientific workflow:

- Document version and libraries (if applicable)

Step 1: File Transfer

- HPC systems operate as “remote” batch-driven systems
  - Interactive use possible, *but* not recommended
- Explicit transfer of files from desktop/laptop to remote system
  - different tools exists: wget, scp, rsync, sftp
  - some remote terminal software includes file-transfer
- DOS/Windows/Linux Line Feed
  - Dos2unix

Live Demo: Tranfering data to the HPC system

- upload the python scripts to the cluster
- download the data to the cluster

Step 1: File Transfer

- HPC systems operate as “remote” batch-driven systems
  - Interactive use possible, *but* not recommended
- Explicit transfer of files from desktop/laptop to remote system
  - different tools exists: wget, scp, rsync, sftp
  - some remote terminal software includes file-transfer
- DOS/Windows/Linux Line Feed
  - Dos2unix

Step 2: General preprocessing

- Shared preparation work applicable for all runs of the problem
  - unpacking,
  - setup of directory structure,
  - clearing of aggregation files, workspace documentation

Live Demo: Setup of directories & launch-script

- unpack the data
- set up a working python environment
- test if the environment works correctly

Step 3: Job-specific preprocessing

- Many applications require case / problem specific preprocessing and setup steps
  - Example: partitioning of the domain according to parallelization
  - Copying input files to the work-directory
  - Downloading specific dataset and preparing computation

Live Demo:

- generation of Input Parameters for each training run
- test if the slurm script was set up correctly

Step 3: Job-specific preprocessing

- Many applications require case / problem specific preprocessing and setup steps
  - Example: partitioning of the domain according to parallelization
  - Copying input files to the work-directory
  - Downloading specific dataset and preparing computation

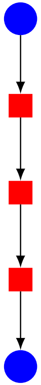
Step 4: Case-specific work

- Job-script and job dependency
  - The usual HPC job

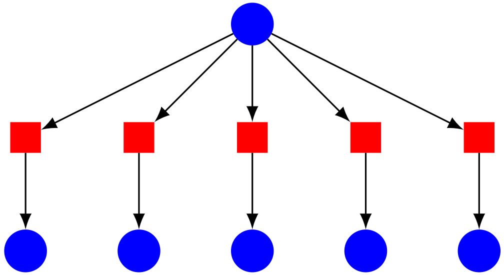
Tangent: Batch-system, Job-Chains, Job-Dependencies

- Batch-systems allows to manage jobs, job-chains, job-steps and job-arrays

Recap: SLURM



Linear workflow with each job depending on its predecessor.



"Trivially" parallel workflow with independent jobs. Work is parallelized "by hand". Oftentimes the the most efficient way to parallelize a workflow.

Recap: SLURM - Job Arrays

Slurm supports a feature called *array jobs*. To make use of this, we need the –array option inside of the jobs script.

- Submitting multiple jobs at once, with a single job script and a single command
- Each array task will reserve the full amount of resources that are requested by the job script (e.g. ——ntasks is the amount of processes that *every* array task can start)
- Each array task has a unique ID, which can be referenced in the job script via the SLURM_ARRAY_TASK_ID variable
- Task IDs can be set explicitly and as a range, with or without a stepsize (see examples)
- The number of maximum tasks that can execute simultaneously can also be set (see examples)

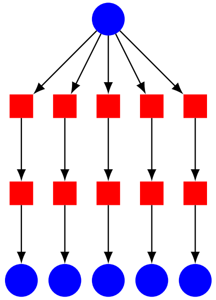
Recap: SLURM - ––

In some workflows jobs depend on each other. Slurm offers the ––dependency option to make jobs relate to others.

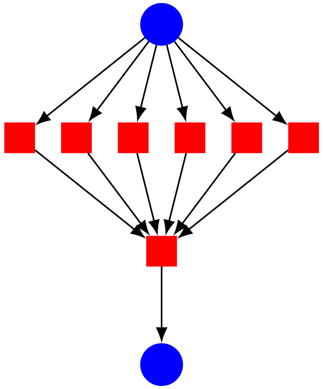
- Set dependencies from other jobs: <type>:<job_ID>
  - "after": Can begin after the specified jobs have **begun execution**
  - "afterany": Can begin after the specified jobs have **terminated**
  - "afternotok": Specified jobs have terminated in some **failed state**
  - "afterok" Specified jobs have **successfully** executed
  - "aftercorr": "After correlated" (later)
  - "singleton": Can begin after **all other jobs with the same job name** have ended

Complex workflows

We can also imagine more complicated workflows with arrays and dependencies.



Linear workflow with each job depending on its predecessor.



Step 5 + 6: General and Job specific postprocessing

Run-specifc postprocessing

- Collect run data,
- Clean-up work-directories
- Extract run-specific data

Problem-specific postprocessing

- Collect data from individual runs
- Process data, i.e. generate figures, graphs or visualization
- Collect & process execution parameters, i.e. accounting information, efficiency . . .

Live Demo:

- Aggregation and visualization of the result data

FAIR Principles

Principles

- **F**indable
- **A**ccessible
- **I**nteroperable
- **R**eusable

Note

The job scripts serves as an important part of the workflow's documentation

Best Practices

- Follow a consistent naming scheme
- Use version control for your scripts as well
- Job script should be self-contained:
  - define all slurm Parameters inside of the job scripts
  - use the long form of slurm parameters (`--cpu-per-task` instead of `-c`)
  - include all used environment modules (with version) or environment variables in job script
- Don't hardcode absolute paths (use a variable instead)
- Add comments if something is not immediately clear
- Include a file explaining the steps necessary outside of the job script

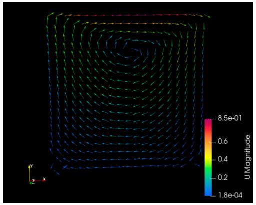
A second workflow example: OpenFOAM

- We will apply the general idea of an HPC workflow to a different use case
- OpenFOAM represents a good example of a 'traditional' HPC problem
- We will skip over (most of) the intricacies of computational fluid dynamics and focus on the workflow steps that are necessary for executing OpenFOAM simulations on an HPC cluster
- Nevertheless, a general understanding of what's happening will be helpful
- The following workflow is adapted from the OpenFOAM tutorial part 2.1, which can be found here:

https://www.openfoam.com/documentation/tutorial-guide/2-incompressible-flow/2.1-lid-driven-cavity-flow#

Some CFD basics

- OpenFOAM's main purpose is to run simulations in the field of *Computational Fluid Dynamics* (CFD).
- CFD applications simulate flows, pressures, velocities, and other properties of macroscopic systems (e.g. simulating air flow around an airplane wing, simulating flow and temperature transport in a cooling circuit, etc.)

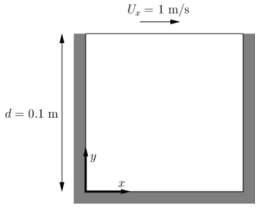


Structure of a CFD simulation

- In general a CFD simulation has a *starting state*, defining the system's boundaries, substance properties (e.g. viscosity), and initial pressure, velocity, etc.
- The simulation then calculates how the system's observables progress and change over time, until reaching an *end state* (either arbitrary or some stable equilibrium)
- Calculations happen on a grid or *mesh*, which partitions the system into parts where the relevant equations can be solved piece-by-piece. Mesh setup and refining are vital parts of preparing simulations.
- More complex systems require more complex meshes, and computational cost for each simulation step scales with the number of mesh points.

A simple CFD use case

- We will be looking at a very simple CFD case: Four walls enclosing a flat square, with three of them stationary and one moving at a constant speed. The moving wall causes pressure and velocity differences in the enclosed substance.



- In OpenFOAM's case, the *preprocessing* step consists of creating and refining the mesh, the *compute* step runs the actual simulation, and *postprocessing* consists of visualizing the simulation and determining whether further refinement is needed.
- The process is *iterative*; we start at a coarse simulation and progressively refine the simulation parameters until a satisfactory result is obtained.

General considerations

Data flow

- CFD simulations, like many 'traditional' simulation tasks, tend to *produce* data.
- That means they start with a moderate (text-form) input and can generate arbitrarily large outputs (depending on simulation length and complexity).

Computational demand

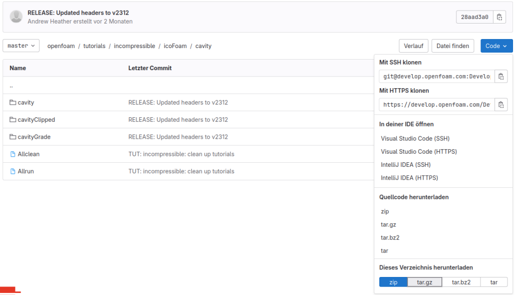
- Preprocessing (mesh generation) and postprocessing (visualization) are comparatively lightweight. Very complex systems might need to move them to a compute node (in the visualization case, maybe to a specialized 'visualization node'), but for smaller cases they can be executed directly on a login node.
- The simulation step constitutes the main computational load and should thus be executed as a batch job - even if the example we show here is very short, any real-world use case will need a significant amount of resources.

Data layout

- OpenFOAM has a data structure that is very common for ‘classical’ simulation applications:
  - Each simulation run has its own subfolder. In OpenFOAM, these are called *cases*.
  - A case contains the run parameters inside the sub-subfolders `system` and `constant`. OpenFOAM will detect these folders and files automatically when given a case folder.
  - Simulation outputs are also written inside the case folder; in OpenFOAM’s case, a new subfolder is created for each timestep that is written out (write interval can be changed in the input files).
- Since we don’t care about the exact contents and syntax of the input, we will download a set of pre-written tutorial files.

Input data acquisition

- The files we need are found at https://develop.openfoam.com/Development/openfoam/-/tree/master/tutorials/incompressible/icoFoam/cavity
- Since we don't want to clone the complete OpenFOAM git, we can use Gitlab's option of downloading subdirectories (click on the 'Code' button, then at the bottom 'Download this directory' as .zip or .tar.gz)



RELEASE: Updated headers to v2312
Andrew Heather erstellt vor 2 Monaten

28aad3a0

master ⌄ openfoam / tutorials / incompressible / icoFoam / cavity

Verlauf | Datei finden | Code ⌄

Name	Letzter Commit
..	
cavity	RELEASE: Updated headers to v2312
cavityClipped	RELEASE: Updated headers to v2312
cavityGrade	RELEASE: Updated headers to v2312
Allclean	TUT: incompressible: clean up tutorials
Allrun	TUT: incompressible: clean up tutorials

Mit SSH klonen
git@develop.openfoam.com:Develo

Mit HTTPS klonen
https://develop.openfoam.com/De

In deiner IDE öffnen
Visual Studio Code (SSH)
Visual Studio Code (HTTPS)
IntelliJ IDEA (SSH)
IntelliJ IDEA (HTTPS)

Quellcode herunterladen
zip
tar.gz
tar.bz2
tar

Dieses Verzeichnis herunterladen
zip | tar.gz | tar.bz2 | tar

Input data transfer

- The downloaded archive will be placed on your local machine.
- Use the techniques we discussed during the other example to transfer the files into your cluster home directory (useable tools: `scp`, `rsync`, FTP clients)
- Once the compressed archive has been copied to the cluster, we'll need to extract the files, either with `unzip` (for .zip) or `tar` (for .tar, .tar.gz and .tar.bz2).

Used software

- We need an OpenFOAM installation that provides the subprograms `blockMesh`, `mapFields`, and `icoFoam`.
- The first two are needed for preprocessing (generating and updating the mesh), while the latter is running the simulation proper ('ico' = 'incompressible'; it is used for incompressible laminar flows, a rather uncomplicated variant of CFD).
- For postprocessing, we need either `paraFoam` (visualizer included in OpenFOAM), or an external `paraview` installation.
- The exact commands needed may vary depending on how OpenFOAM is installed (built from scratch or provided as a container).
  - E.g.: In case of MaRC3 (Marburg), all OpenFOAM commands are prefixed with `openfoam`, which starts the container (i.e. `openfoam blockMesh`, `openfoam icoFoam`, etc.)

ParaView

- The ParaView interface can seem somewhat overwhelming and obscure at first, since it has many incredibly detailed options that are only useful for domain experts.
- The main principle is opening a data source (in our case a `*.foam` file in the case folder) and applying ‘filters’ to it to visualize aspects of this data.
- A full ParaView tutorial would be beyond the scope of this course; for a few ideas on what to visualize, we can refer to paragraph 2.1.4 in

https://www.openfoam.com/documentation/tutorial-guide/2-incompressible-flow/2.1-lid-driven-cavity-flow#

First workflow iteration: Preprocessing

- As a first step, we will generate a simple 2D 20x20 mesh to run the simulation.
- Since the case is very small, we can do this without submitting it as a batch job
  - However, it's a good practice for setting up a simple HPC job!
- To generate the mesh, we need to run the following command inside the `cavity/cavity/` directory:

```
blockMesh
```

- Remember that we can run the command from anywhere in the file tree by adding a `-case <path>` parameter:

```
blockMesh -case /home/<username>/<path_to_cavity_folder>
```

First workflow iteration: Compute

- The simulation proper is run by calling `icoFoam` from inside the case directory, or `icoFoam -case <path>` from anywhere.
- Although it is not strictly needed since the calculation will only take a few seconds, we nevertheless should submit it as a batch job - any real-world use case would require it.
- The resource requirements for this job are minimal (1 CPU core, 4GB of memory, and 10 minutes of runtime should suffice)
- You can either store job files inside the case directory, or collect batch job inputs and outputs in a different folder. What are the advantages and disadvantages of either?

First workflow iteration: Postprocessing

- The postprocess step in OpenFOAM consisty of visualizing the simulation and checking whether further refinement might be beneficial.
- Visualization is done with ParaView. If the OpenFOAM installation includes the `paraFoam` module, we simply need to start the tool from inside the case folder (check if X-Forwarding is switched on in your SSH session!).
- With external ParaView, we need to generate a dummy file in the case directory and then load it:

```
# Inside the case folder:
touch cavity.foam
paraview cavity.foam
```

Second workflow iteration: Preprocessing 1 - New mesh

- The mesh for the first simulation is rather coarse. We'll generate a second finer mesh (double resolution), map the final step of the previous iteration onto it, and prepare to simulate further steps with the new resolution.
- In OpenFOAM's logic, this marks a new *case*. First of all, we copy the results from last iteration into a new subfolder:

```
cp -r cavity cavityFine
```

- In the new `cavityFine` case folder, we need to edit the file `./system/blockMeshDict` to change the mesh resolution; find the line starting with `hex` and change `(20 20 1)` to `(40 40 1)`. This effectively doubles the resolution along the x and y axis.
- We can now generate the new mesh:

```
blockMesh
```

Second workflow iteration: Preprocessing 2 - Mesh mapping and simulation steps

- We want to map the new mesh onto the last step of the previous simulation; to communicate this to OpenFOAM, we open `system/controlDict` and change the line for `startTime` from `0.0` to `0.5` (i.e. to the last step of the previous run) and delete the numbered folders (`0`, `0.1`, `...`, `0.5`) (we'll recreate this folder while mapping).
- Then, we let OpenFOAM map the new mesh onto the old result:

```
mapFields ../cavity -consistent
```

- Further changes to `system/controlDict`:
  - Change `deltaT` from `0.005` to `0.0025` (a finer resolution needs smaller time steps for numerical reasons)
  - Change `endTime` from `0.5` to `0.7` (otherwise startTime would be identical to endTime)
  - Set `writeControl` to `runTime` and `writeInterval` to `0.1` (keeps the output interval at 'every 0.1 seconds' with new step size).

Second workflow iteration: Compute

- The only thing that has changed for the compute step is the path to the case folder.
- There is no need to increase the amount of compute resources; if you have a jo0b script for the first simulation, adapting it should be trivial.
- As before, the simulation proper is run (ideally from within a compute job) with `icoFoam [-case <path_to_cavityFine_folder>]`

Second workflow iteration: Postprocessing

- Postprocessing is, again, mostly checking various simulation properties with ParaView.
- OpenFOAM includes tools to plot a wide range of system properties, which is fittingly called `postProcess`
- As an example, we will show how to plot overall magnitude of velocity vectors, and directional components of velocity:

```
postProcess -funcs '(components(U) mag(U))'
```

- This data is then available in ParaView for plotting. The filter for this is called ‘Plot over line’; the line coordinates should be set to `(0.05, 0, 0.005)` and `(0.05, 0.1, 0.005)` for *Point1* and *Point2*, and resolution can be set to 100.

Third workflow iteration: Preprocessing 1 - Graded mesh

- In CFD it is common to have different regions in a system, where some undergo big changes while others barely change during a simulation.
- Therefore, meshes are often not completely uniform - instead, they are defined to be finer in 'interesting' regions and allowed to be coarser in 'uninteresting' regions. These are called *graded meshes*.
- The case folder `cavityGrade` includes a starting configuration for a graded mesh.
- At first, we will again generate the mesh:

```
cd ../cavityGrade
blockMesh
```

- We can also open the case folder in ParaView (don't forget to generate a `.foam` file first if using a standalone ParaView installation) to check the newly generated mesh.

Third workflow iteration: Preprocessing 2 - Mesh mapping

- We will first edit `system/controlDict` to set starting and ending point for the new simulation:
  - Set `startTime` to `0.7` (the last step of the cavityFine run)
  - Set `endTime` to `0.8` (for our simple system, we don't need to run for longer to achieve a stable state)
- Then, we map the last step of the cavityFine case onto the graded mesh, which we will use as the starting point of the third simulation:

```
mapFields ../cavityFine -consistent
```

Third workflow iteration: Compute

- As in the second iteration, only the case folder has changed, so we can easily adapt our previous job script to run the 'cavityGrade' case
- Once again, the `icoFoam` module will be used.
- By now, you should see the repeating pattern of this workflow. With some preparation, the predictable folder layout and the submit scripts offer opportunities for automation. Can you think of some?

Third workflow iteration: Postprocessing

- The results can again be visualized in various ways with ParaView.
- This refinement cycle will go on longer for real use cases - stay aware of the fact that we are *generating* data! That means:
  - While old simulations are used for mapping (and also for reference, if a run needs to be recreated), they do not need to stay on fast storage indefinitely
  - If immediate and fast access to old data is no longer needed, it should be archived in a feasible way (compressed, moved to 'cold storage', etc.)
  - Text-only data such as the simulation input files can be nicely stored in a version control system (e.g. git), which also means that the whole edit history is kept and any previous version of the files is readily available.
  - For the output data, care should be taken to include relevant metadata, so that each result set can be readily matched to input file versions, software versions, author, etc.
    - ★ Apply *FAIR* principles!

Conclusion

- Having walked through two very different HPC workflows and looking back at the general principles presented at the start of the course, it should be obvious that even very different use cases follow similar patterns:
  - Initial considerations about data layout and data transfer
  - Preprocessing, compute and postprocessing, often in multiple iterations
  - Considerations of how, where, and what to save of the results
- In a generalized form, these patterns can be applied to any HPC use case. What these steps may contain in detail is filled out by domain knowledge, experimenting, and experience.
- Ultimately, the idea of a generalized HPC workflow offers guidance on *what questions to ask*, since the actual answers will be different for every use case.

Image sources

- Slide 20: http://ufldl.stanford.edu/housenumbers
- Slide 41: https://www.openfoam.com/documentation/tutorial-guide/2-incompressible-flow/2.1-lid-driven-cavity-flow#
- Other images: Own work, © 2024 Competence Center for High Performance Computing in Hessen (HKHLR)