

Onboarding

Julia on HLRS Laptop/Cluster

HLRS Laptops

Most of the course can be completed on the laptops.

- Equipped with NVIDIA GPU
- Jupyter + VS Code
 - jupyter lab
 - code
- Course materials
 - [\\$HOME/JuliaHLRS24](#)

Let's start Jupyter!

→ `cd JuliaHLRS24`

→ `jupyter lab`

→ `notebooks/Day1/1_julia_fundamentals.ipynb`

HLRS Training Cluster

The cluster training.hlr.de has two types of nodes.

CPU nodes

- “skl”
- 2x Intel Skylake
- 40 cores total

GPU nodes

- “clx-ai”
- 2x Intel Cascade Lake
- 36 cores total
- 8x NVIDIA V100

Jobs are scheduled with PBS Pro.

- Submit a job:
 - `qsub job_script.sh`
- Check on your queued/running jobs:
 - `qstat -nw`

VS Code → HLRS Cluster

Run VS Code on a cluster node via SSH.

Login node

- Works fine, just connect to
 - `accountname@training.hlrs.de`

Compute node

- At HLRS, possible but inconvenient
 - `SetEnv PBS_JOBID=...`
 - `SSH ProxyJump`

We will stay on Login nodes for the course.

To get Julia, load the necessary system modules.

- Modules on the HLRS training cluster
 - `module use julia`
 - `module use nvidia/nvhpc` # MPI+CUDA
 - `module use compiler/nvidia` # MPI+CUDA
- Outside of the course: If there is no (working) system module, use standard binaries provided by [juliaup](#).

Comment: Julia depot is on the parallel file system.

- Julia depot = where Julia stores stuff
 - packages
 - binary dependencies
 - ...
- Environment variable: JULIA_DEPOT_PATH
- Why not \$HOME?
 - Quotas
 - Read-only from compute jobs (sometimes)

Julia VS Code integration via extension.

The screenshot displays the VS Code interface with the Julia extension installed. The Explorer sidebar on the left shows the project structure, including a `JULIAINTERPRETER` section with files like `.github`, `.vscode`, `benchmark`, `bin`, `deps`, `docs`, `src`, `test`, `.gitignore`, `.travis.yml`, `appveyor.yml`, and `LICENSE.md`. Below this is the `JULIA EXPLORER: JULIA WORKSPACE` section, which lists the project's dependencies and the `ans` package. The Outline sidebar on the left shows the `mandel` function and its associated variables (`c`, `maxiter`, `n`, `z`, `i`, `x`). The main editor window shows the `mandel` function definition in `Untitled-1`. The function is a generic function with one method, `mandel`, which takes a collection `c` and a maximum iteration count `maxiter`. It iterates over the elements of `c` and prints the iteration count for each element. The function is defined as follows:

```
9  ...end
10  ...return maxiter
11  end mandel (generic function with 1 method)
12
13  for i in 1:10
14  ...println(i)
15  end
16
17  map
```

The `map` function is highlighted, and a tooltip is displayed, showing the function signature `map(f, c...) -> collection` and a description: "Transform collection `c` by applying `f` to each element. For multiple collection arguments, apply `f` elementwise." The tooltip also includes an example: `julia> map(x -> x * 2, [1, 2, 3])` resulting in a 3-element `Array{Int64,1}` with the value `2`.

The Julia REPL (REPL) is visible at the bottom of the editor, showing the prompt `julia>`. The output of the `mandel` function is displayed in the REPL, showing the iteration count for each element of the collection `c`.

On the right side of the editor, a plot titled "Julia Plots (2/2)" is shown. The plot displays a red curve, labeled `y1`, which represents the output of the `mandel` function. The x-axis ranges from 4 to 8, and the y-axis ranges from 0.0 to 1.0. The curve starts at approximately (4, 0.0), reaches a minimum of about 0.1 at x=5, and then rises to a maximum of about 1.0 at x=8.

On the cluster, the extension requires a wrapper.

- [Julia: Executable Path](#) should point to a wrapper script, like this one:

```
#!/bin/bash
[...]  
  
# Load modules  
module load julia  
module load nvidia/nvhpc  
module load compiler/nvidia  
  
# Act like Julia (i.e. pass on all arguments)  
exec julia "${@}"
```

Let's do it!

→ `exercises/Day1/1_cluster_onboarding`

