Best practices for compiling software and local Python/R/… installations

KU Leuven HPC support / ICTS

Wouter Van Assche
Maxime Van den Bossche

VLAAMS SUPERCOMPUTER CENTRUM | *Innovative Computing for A Smarter Flanders*

Vlaanderen is supercomputing

vscentrum.be

Overview

1. When (not) to go for local installations
2. Know your application
   - Software bill of materials
   - Performance profile
3. Considerations at the package level
   - Dependencies
   - Compilers and compiler options
   - Containers
   - Install locations
4. Considerations at the stack level
   - Frameworks for managing local installations
   - Dealing with heterogeneous hardware
5. Python- and R-specific considerations
   - Local installation methods
   - Compatibility with Open OnDemand apps
6. Hands-on / Questions

Central versus local installations

When (not) to go for local installations

Central installations (`/apps/leuven/...` modules):

= preferred approach unless the software:

- is either in active development,
  (package updates, bugfixes, features, …)
- or only consists of interpreted code.

python™ R Perl …

→ Local installation can be more convenient

Avoid spending large amounts of compute resources with interpreted code

example: Python lists versus NumPy arrays

VLAAMS SUPERCOMPUTER CENTRUM

Know your application

Know your application

Software bill of materials (SBOM)

- What are the major components and dependencies?
- How are these compiled/interpreted/provided/…?
  What would be the alternatives?

Performance profile

- Where is the most CPU-time and (if applicable) GPU-time being spent?

Know your application

<u>Example</u>: electronic structure calculations with xTB through a Python interface (ASE)

Component		
Runscript		
└── self (Python)		
└── calls to ASE		
└── self (Python)		
└── calls to NumPy & SciPy		
└── self (Python, C/C++)		
└── calls to BLAS		
└── calls to xTB		
└── self (Fortran, OpenMP)		
└── calls to (threaded) LAPACK		

Know your application

<u>Example</u>: electronic structure calculations with xTB through a Python interface (ASE)

Component	Performance critical?	
Runscript	N	
└── self (Python)	N	
└── calls to ASE		
└── self (Python)	N	
└── calls to NumPy & SciPy		
└── self (Python, C/C++)	N	
└── calls to BLAS	N	
└── calls to xTB		
└── self (Fortran, OpenMP)	Y	
└── calls to (threaded) LAPACK	Y	

Know your application

<u>Example</u>: electronic structure calculations with xTB through a Python interface (ASE)

Component	Performance critical?	Influential choices
Runscript	N	┐
└── self (Python)	N	┤
└── calls to ASE		├── none (only interpreted code)
└── self (Python)	N	┘
└── calls to NumPy & SciPy		
└── self (Python, C/C++)	N	choice of C/C++ compiler & compiler options
└── calls to BLAS	N	choice of provider (MKL, OpenBLAS, BLIS, …)
└── calls to xTB		
└── self (Fortran, OpenMP)	Y	choice of Fortran compiler & compiler options
└── calls to (threaded) LAPACK	Y	choice of provider (MKL, OpenBLAS, BLIS, …)

Considerations at the package level

Package level: dependencies

Performance depends on implementation, hardware, parallelism, …

Dependency	Providers (CPU)	Providers (GPU)
BLAS / LAPACK	MKL, OpenBLAS, (AOCL-)BLIS, …	cuBLAS/cuSOLVER, hipBLAS, …
PBLAS / ScaLAPACK	MKL, (AOCL-)ScaLAPACK, …	
MPI	Open MPI, MPICH, Intel MPI, …	
FFTW	(AOCL-)FFTW, MKL, …	cuFFT, hipFFT, …

Not always immediately clear how dependencies are satisfied
(OpenBLAS in NumPy wheels from PyPI, MPICH in mpi4py from conda-forge, …)

Beware of local MPI installations (possible misconfiguration)
We recommend to use centralled installed modules (OpenMPI, impi)

Package level: compilers and compiler options

✓ enable optimizing transformations (`-O…` and other options)

✓ allow to use all available instructions on the host device(s):

CPU: `-march=native` (GCC) / `-xHost` (Intel) / `-Xcompiler="..."` (NVIDIA)

GPU: `--gencode=arch=compute_xx,code=sm_xx` (NVIDIA)

P100/V100/A100/H100: `xx=60/70/80/90`

Optimizing for multiple architectures in one “fat” binary
= non-trivial and is not supported by all compilers

MKL, CUDA, .. libraries do detect the CPU/GPU model and dispatch accordingly
OpenBLAS as well (but not in the centrally installed OpenBLAS modules)

Package level: compilers and compiler options

✓ enable optimizing transformations (`-O…` and other options)

✓ allow to use all available instructions on the host device(s):

CPU: `-march=native` (GCC) / `-xHost` (Intel) / `-Xcompiler="..."` (NVIDIA)

GPU: `--gencode=arch=compute_xx,code=sm_xx` (NVIDIA)

P100/V100/A100/H100: `xx=60/70/80/90`

Use Slurm jobs to produce host-CPU/GPU-optimized binaries

Try to use the most recent available compiler (e.g. 2023a modules)
especially for newer hardware (e.g. AMX instructions on Sapphire Rapids)
Intel: switch to oneAPI compilers (icx, ifx, …)

Package level: containers

All considerations so far also apply to containers

Available platforms: APPTAINER podman

Other pros & cons:

✓ control over OS dependencies

✓ some software can be easier to install with OS package managers (e.g. GUI or legacy apps)

± compute/memory/disk overheads are normally acceptable (if not negligible)

✗ MPI is supported but requires careful setup for good performance

✗ generally not possible to use centrally installed modules inside container

? reduced IO load on parallel file systems (?)

Package level: containers

All considerations so far also apply to containers

Available platforms: APPTAINER podman

- Use Dockerfiles/Apptainer definition files/… (avoid "`docker commit ...`")
- Specify the base image version (e.g. "`FROM rockylinux:8.9`")

Where to find more:

https://docs.vscentrum.be/software/singularity.html

https://docs.docker.com/build/building/best-practices

https://hpcleuven.github.io/artifactory-doc

Package level: install locations

${VSC_HOME}	✗	only intended for configuration files; only 3 GB quotum
${VSC_SCRATCH}	✗	periodic cleaning of unused files
${VSC_DATA}	✓	preferred location
staging	✓	beware: single-copy; less well suited for small files

VLAAMS
SUPERCOMPUTER
CENTRUM

Considerations at the stack level

Stack level: frameworks for managing local installations

- General-purpose tools with focus on building HPC software from source: (EasyBuild or Spack) + Lmod

    ✓ support for many different installation procedures

    ✓ good control over dependencies, build options, …

    ✗ there is some learning curve

    ✗ building from source can take a while (tip: make use of centrally installed modules)

- Focused on (binary, self-contained) Python & R packages: (mini) CONDA , Mamba

    ✓ quick to set up and use

    ✗ application performance may be suboptimal

    ✗ installations can be slow on shared filesystems due to many metadata queries

Stack level: dealing with heterogeneous hardware

- Predefined environment variables:

Cluster	Partitions	${VSC_ARCH_LOCAL}
Genius	bigmem, gpu_p100, interactive, login nodes, superdome	skylake
Genius	batch, gpu_v100	cascadelake
Genius	amd	naples
wICE	batch, bigmem, gpu_a100, hugemem, interactive	icelake
wICE	batch_sapphirerapids	sapphirerapids
wICE	gpu_h100	zen4

Other variables: `${VSC_ARCH_SUFFIX}` (`-h100` on gpu_h100, empty everywhere else)
`${VSC_OS_LOCAL}` (e.g. `rocky8`)
`${VSC_INSTITUTE_LOCAL}` (e.g. `leuven`)
`${VSC_INSTITUTE_CLUSTER}` (e.g. `genius`, `wice`)

Stack level: dealing with heterogeneous hardware

- Example directory structure:

```
${VSC_DATA}
└── apps
    └── ${VSC_INSTITUTE_LOCAL}
        └── ${VSC_OS_LOCAL}
            └── ${VSC_ARCH_LOCAL}${VSC_ARCH_SUFFIX}
                └── 20XXx
```

e.g. `${VSC_DATA}/apps/leuven/rocky8/sapphirerapids/2023a/...`

Python- and R-specific considerations

Python & R: local installation methods

(a) Centrally installed modules + local environment

Python: venv, R: local libraries

✓ leverage already (optimally) installed packages

✗ only possible to install Python/R packages

(b) Conda-like frameworks

✓ also possible to install interpreters, libraries, …

✗ more user-installed packages ⇒ more occasions for suboptimal performance

Reclaim space in your ${VSC_DATA} by cleaning package caches
(“`conda clean --all`”, “`pip cache purge`”)

Python & R: local installation methods

(a) Centrally installed modules + local environment

Python: venv, R: local libraries

✓ leverage already (optimally) installed packages

✗ only possible to install Python/R packages

Basic example (Python):

```
module load Python/3.10.8-GCCcore-12.2.0
cd ${VSC_DATA}
python -m venv ./venv_mytools
source ./venv_mytools/bin/activate
pip install numpy==1.26.4  # note: OpenBLAS
```

Python & R: local installation methods

(a) Centrally installed modules + local environment

Python: venv, R: local libraries

✓ leverage already (optimally) installed packages

✗ only possible to install Python/R packages

In this case we recommend creating different installations for different CPU architectures

Basic example (R):

```
module load R/4.2.2-foss-2022b
# Tip: also consider R-bundle-Bioconductor (available for R >= 4.2.2)
cd ${VSC_DATA}
mkdir -p Rlibs/${VSC_ARCH_LOCAL}/R-${EBVERSIONR}
export R_LIBS_USER=${VSC_DATA}/Rlibs/${VSC_ARCH_LOCAL}/R-${EBVERSIONR}
R
> .libPaths()
> install.packages("mlr3")
```

Python & R: local installation methods

(b) Conda-like frameworks

✓ also possible to install interpreters, libraries, …

✗ more user-installed packages

⇒ more occasions for suboptimal performance

(mini) CONDA = mature, but dependency resolution can be slow ⇒ Mamba (drop-in replacement)

Consider using the `intel` conda channel for MKL and other libraries
https://software.intel.com/content/www/us/en/develop/articles/using-intel-distribution-for-python-with-anaconda.html

Python & R: compatibility with Open OnDemand apps

Install your own kernels for use in the JupyterLab OOD app

Python (virtual or Conda environment):

```
source activate myenv
pip/conda install jupyter_client
python -m ipykernel install --user --name=myenv --display-name=myenv
```

R (currently only via Conda environments)

```
source activate myenv
conda install jupyter_client r-irkernel
Rscript -e 'IRkernel::installspec(name="myenv", displayname="myenv")'
# Add the following to your ~/.bashrc file:
export XDG_DATA_HOME="$VSC_DATA/.local/share"
```

Python & R: compatibility with Open OnDemand apps

RStudio Server: currently uses a system-installed R interpreter

- no preinstalled packages
- only few OS dependencies available

R (Conda environment):

Not supported

R (local R libraries):

Current setup is (too) limited due to the system-installed R interpreter

Future OOD releases will use module-based R installations for RStudio

For the time being we recommend to use JupyterLab instead of RStudio Server

Documentation links

https://docs.vscentrum.be/software/software_development.html

https://docs.vscentrum.be/software/python_package_management.html

https://docs.vscentrum.be/software/r_package_management.html

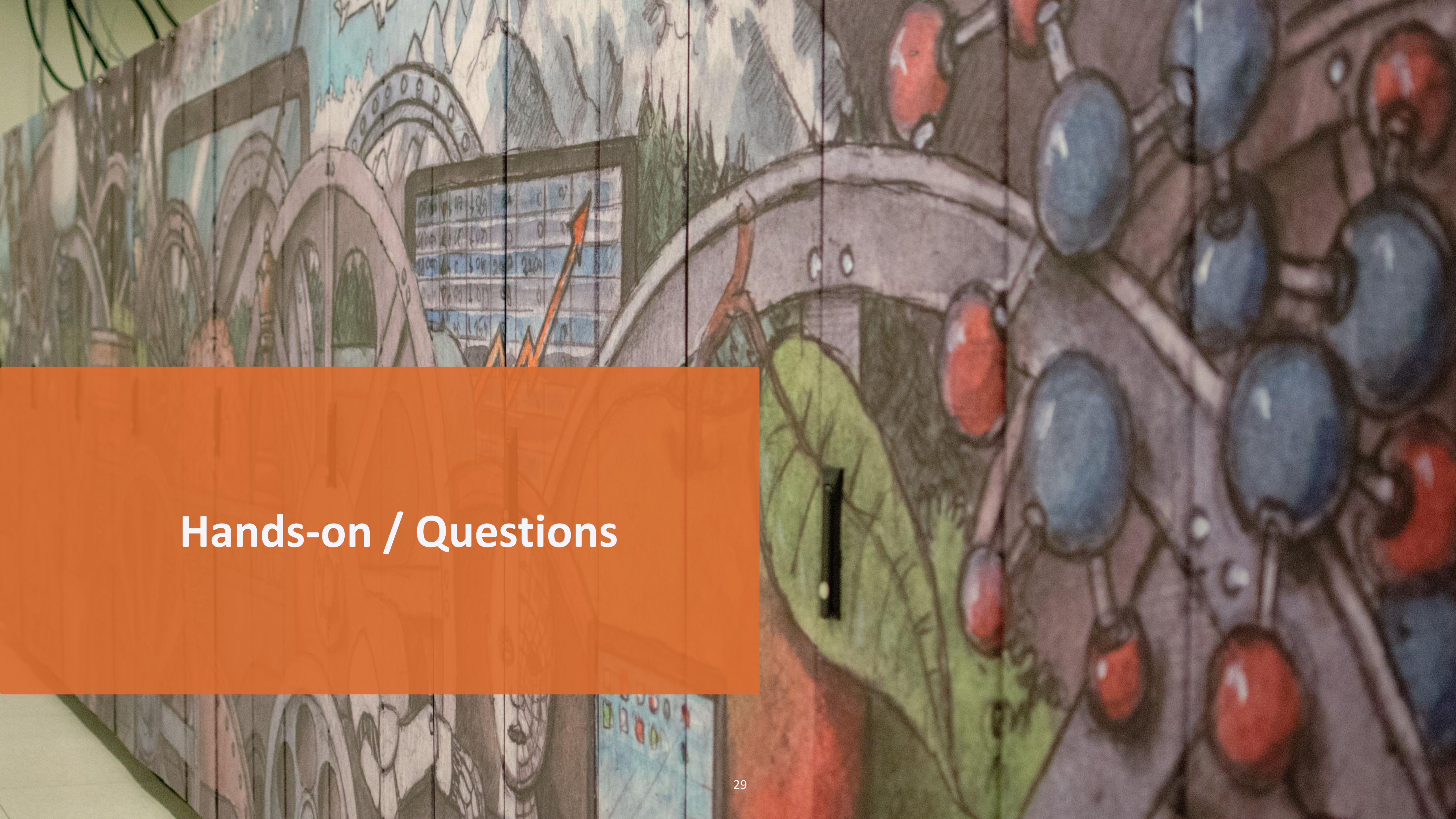
https://docs.vscentrum.be/leuven/wice_advanced_guide.html#compiling-software

https://docs.vscentrum.be/en/latest/software/singularity.html

https://docs.vscentrum.be/leuven/genius_2_rocky.html#the-cluster-module

https://hpcleuven.github.io/artifactory-doc

https://github.com/hpcleuven/VSC_USER_DAY_2024



Hands-on / Questions