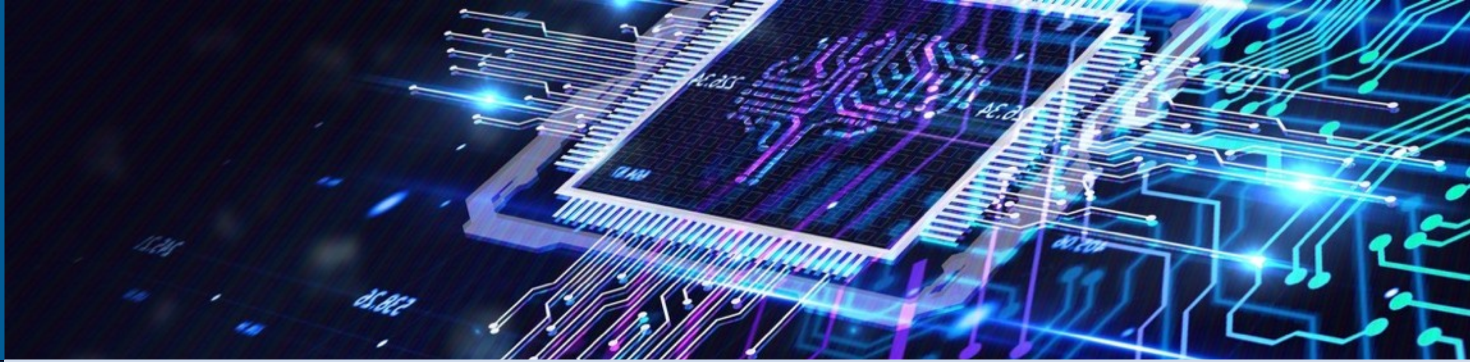
CSC

ICT Solutions for Brilliant Minds



Reducing File System Stress Caused by Large Python Installations Using Containers

Henrik Nortamo, CSC-IT Center for Science

CUG2023, 10.5.2023

Agenda

- Our issue with python
- Some pseudo benchmarks
- Our solution
- How it works
- Limitations
- Future work

Why Python causes issues

- Python environments for some user groups tend to grow very large.
  - Python accesses a lot of files, even with a few imports
- Higher expectations on interactivity
  - REPL
  - Iteration
  - Web interfaces place the user in a different context
- Lustre generally does not deal well with a large number of small files
  - Both for individual users and global impact
  - Hard limits imposed by quotas

```
$ find my_python_installation | wc -l
423712
```

“The performance with small files will not be optimal”

“Accessing small files on the Lustre filesystem is very inefficient”

“The Lustre file system is the worst place to store a lot of small files”

Quotes found in technical documentation from Aalto University, INCD and ETH zürich

Python in a container

- Using containers is the obvious solution unless you want to redesign Python or force your users to switch to another language
  - The container image is a single file from the point of view of Lustre
- Installing and running Python environments from a container is nothing new
- **However**, some use cases become much harder or are blocked entirely
  - MPI bindings, workflow managers, integration into existing pipelines, extending the installation
  - Containers need to be built off premise***** → extra steps for end-users

Target: Create an easy way for users to containerize their Python installation and enable as many use cases as possible

***** *Newer version of singularity/apptainer will allow you to build with fakeroot + sandbox, but not from recipes*

Some additional background

- Conda → package management system mainly used for python
- No usernamespaces on LUMI, or the Finnish national systems
  - singularity CE and apptainer running in SUID mode.
- No squashfuse / fusermount commands on LUMI
- Everything presented here done on Lustre

Duration of imports

Importing 7 python packages from a relatively sizable conda installation

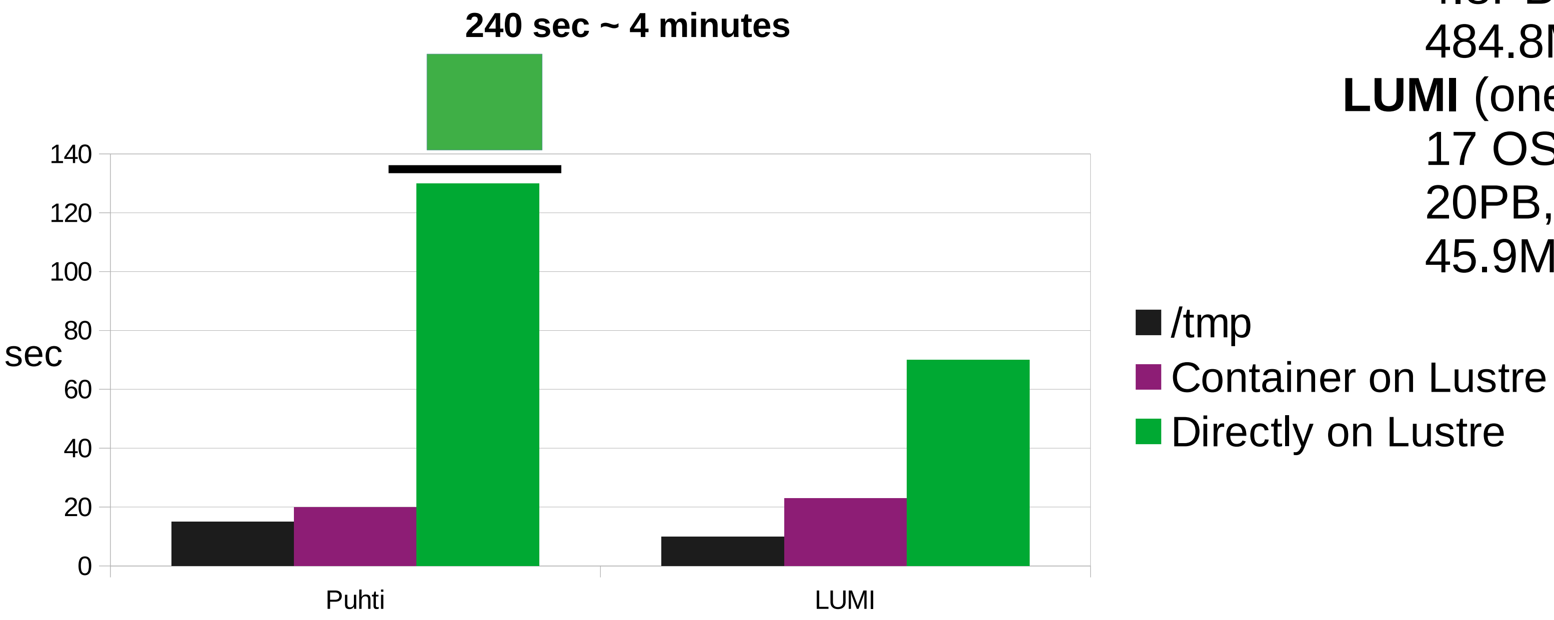
```python
import leafmap
import pdal
import dask
import geopandas
import scipy
import laszip
import hvplot
print("Hello CUG2023")
```

→ 8K `fstat` calls and 7K `read` system calls

Environment is based on a geocomputing installation provided at CSC


```yaml
- boto3
- bottleneck
- cartopy
- cfgrib
- copc-lib
- dask
- dask-geopandas
- dask-jobqueue
- dask-labextension
- dask-ml
- descartes
- earthengine-api
- geoalchemy2
- geopandas
- geopy
- graphviz
- gstools
- hvplot
- imageio
- imbalanced-learn
- jinja2
- jupyterlab
- jupyterlab-git
- keplergl
- laspy
- laxpy
- leafmap
- lidar
- lxml
- metpy
- natsort
- ncview
- netcdf4
- networkx
- osmnx
- owslib
- pdal
- pip
- plotly
- psy-view
- py6S
- pydeck
- pygmt
- pyicu
- pyntcloud
- pyogrio
- pyproj
- pysal
- pystac-client
- python-cdo
- python-graphviz
- python-igraph
- python-keystoneclient
- python-pdal
- python-swiftclient
- python-wget
- rasterio
- rasterstats
- rioxarray
- rtree
- satpy
- scalene
- scikit-image
- scikit-learn
- sentinelsat
- shapely
- spyder
- stackstac
- tifffile
- wrf-python
- xarray
- xarray-spatial
- xarray_leaflet
- xesmf
- pip:
  - laszip
  - open3d
  - git+https://github.com/ernstste/landsatlinks.git
  - git+https://github.com/pangeo-data/xesmf.git
```

Duration of imports

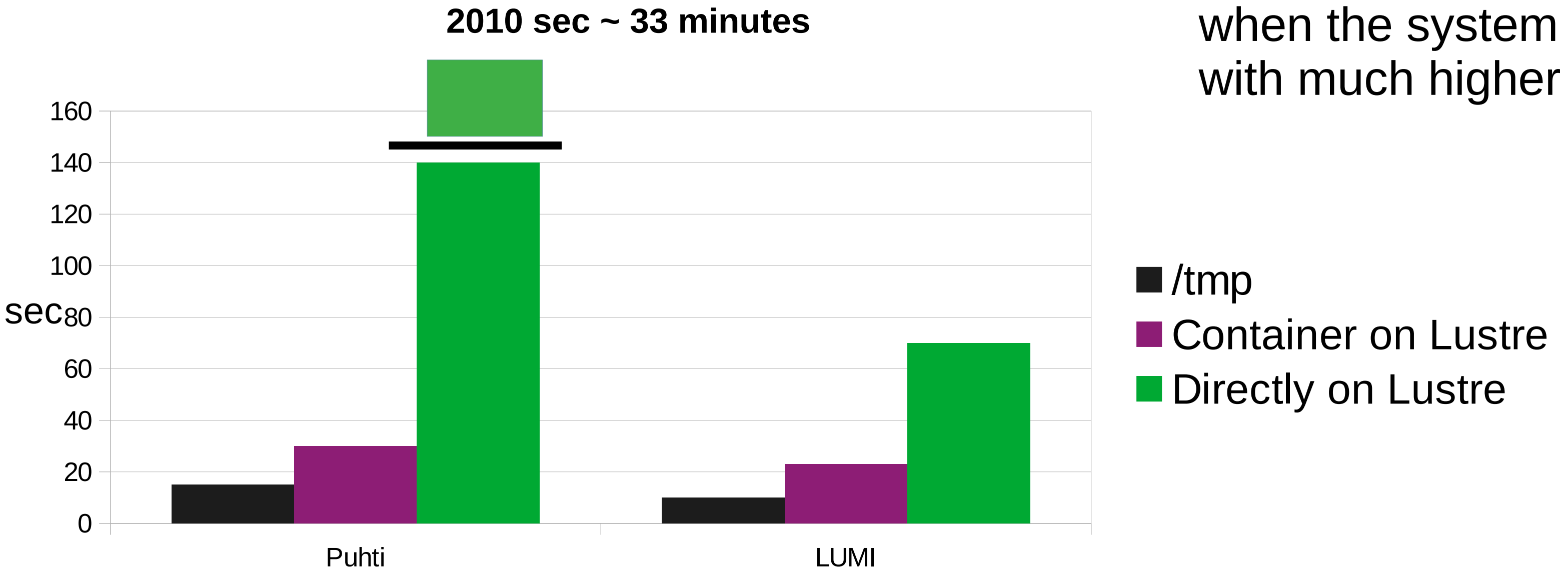


Puhti:
- 8 OSS
- 4.8PB, 57% used
- 484.8M Inodes used

LUMI (one filesystem)**:**
- 17 OSS
- 20PB, 9% used
- 45.9M Inodes used

Benchmarks done on live systems → very noisy

Duration of imports

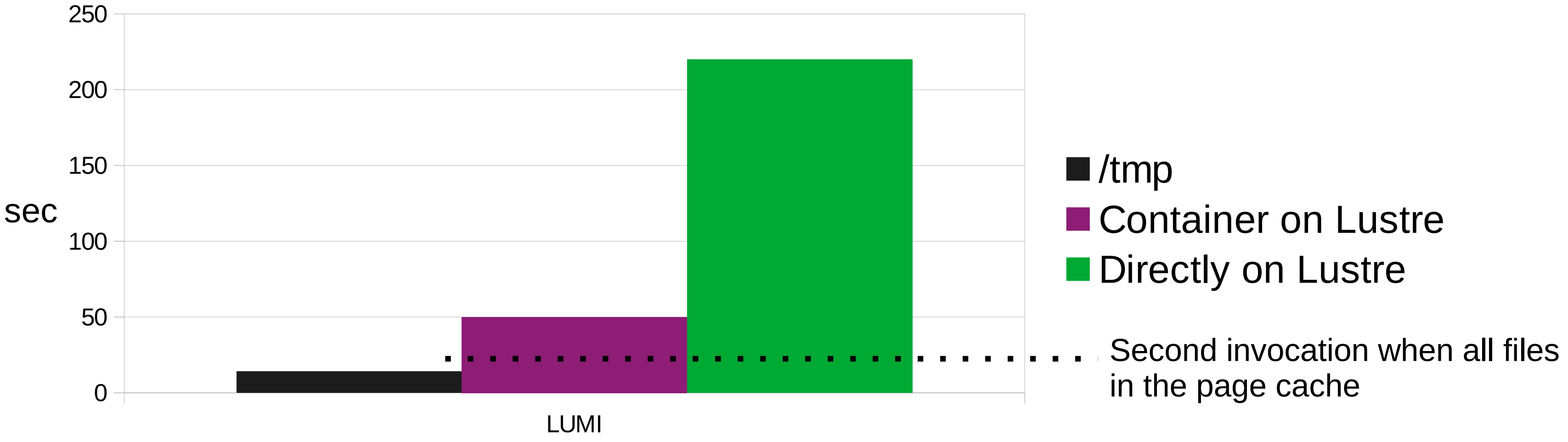


What will this look like for LUMI when the system is at full load with much higher disk usage?

When Puhti was under extremely heavy load + some users were doing less than nice things to the filesystem

Duration of imports

Stress test, instead of 7 import we have 72



→ 22K `fstat` calls and 17K `read` system calls

Our tool

- **Tykky** (https://github.com/CSCfi/hpc-container-wrapper) installs the user’s Python environments into a container, and then generates a set of wrappers which try to hide the container as much as possible
  - Separate modes of operation for creating conda installations and installations based on a virtual environment
- In production use, by end users and CSC staff

```
$ conda-containerize new --prefix MyEnv env.yml
[ INFO ] Constructing configuration
[ INFO ] Using /tmp/nortamoh/cw-VQOSFK as temporary directory
[ INFO ] Fetching container docker://opensuse/leap:15.4
[ INFO ] Running installation script
[ INFO ] Using miniconda version Miniconda3-latest-Linux-x86_64
[ INFO ] Installing miniconda
```

env.yml

```
channels:
  - conda-forge
dependencies:
  - numpy
```

Reduction from 400K files to 2K files for our example case

What it looks like

```
$ ls MyEnv/
_bin  bin  common.sh  container.sif  img.sqfs  share
$ ls MyEnv/bin/ | head
2to3
2to3-3.10
acountry
acyclic
adig
aec
ahost
annotate
aomdec
aomenc
```

```
$ ls MyEnv/
_bin  common.sh      img.sqfs  lib64       share
bin   container.sif  lib       pyvenv.cfg
$ ls MyEnv/bin/
_debug_exec   pip   pip3.9  python3
_debug_shell  pip3  python  python3.9
```

bin/python

```bash
#!/bin/bash

export OLD_PATH=$PATH
export PATH="/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/bin"
SOURCE="${BASH_SOURCE[0]}"

_O_SOURCE=$SOURCE

while [ -h "$SOURCE" ]; do # resolve $SOURCE until the file is no longer a symlink

  DIR="$( cd -P "$( dirname "$SOURCE" )" >/dev/null 2>&1 && pwd )"

  SOURCE="$(readlink "$SOURCE")"

  [[ $SOURCE != /* ]] && SOURCE="$DIR/$SOURCE" # if $SOURCE was a relative symlink, we need to resolve it relative to the path where the symlink file was located
done

DIR="$( cd -P "$( dirname "$SOURCE" )" >/dev/null 2>&1 && pwd )"



source $DIR/../common.sh

if [[ ${_CW_IN_CONTAINER+defined} ]];then
    export PATH="$OLD_PATH"
    exec -a $_O_SOURCE $DIR/../_bin/python "$@"
else

        if [[ ( -e $(/usr/bin/dirname $_O_SOURCE )/../pyvenv.cfg && ! ${CW_FORCE_CONDA_ACTIVATE+defined} ) || ${CW_NO_CONDA_ACTIVATE+defined} ]];then
        export PATH="$OLD_PATH"
        /usr/bin/singularity --silent exec  $DIR/../$CONTAINER_IMAGE bash -c " exec -a $_O_SOURCE $DIR/python $( test $# -eq 0 || printf " %q" "$@" )"
    else
        export PATH="$OLD_PATH"
        /usr/bin/singularity --silent exec  $DIR/../$CONTAINER_IMAGE  bash -c "exec -a $_O_SOURCE $DIR/python $( test $# -eq 0 || printf " %q" "$@" )"
    fi
```

Installation

1) Launch a base container matching the host operating system
   - **Mount all top level paths from the host**
   - Mount some local disk or `/tmp` to `/LUMI_<random_hash>`
2) Install miniconda to `/LUMI_<random_hash>`
   - Create environment based on user input
   - Run any extra user commands

Now outside the container!

3) **Create squashfs filesystem image from content of `/LUMI_<random_hash>`**
4) **Generate wrappers for all executables in the installation**

By mounting the full host filesystem, we can utilize all installed software e.g. the whole Cray module stack

Running

1) User calls the wrapper the same way the use a normal installation: `MyEnv/bin/python3`
   - In practice drop in replacement for a lot of scripts
   - Wrapper handles propagating host environment and variables into the container
   - Wrapper handles invocation if it is already inside a container
2) Launch a base container matching the host operating system
   - **Mount all top level paths from the host**
   - **Mount some squashfs image to `/LUMI_<random_hash>`**
3) Execute the actual program inside the container
   - If installation is conda based, activate the conda environment
   - Edit the zeroth argument on execution

Running, some examples

- **mpi4py**
  - `pip-containerize new --prefix MyEnv/ req.txt`
  - `srun -n 2 -N 2  MyEnv/bin/python3 osu_latency.py`
- **Dask**
  - The correct absolute path to the interpreter is inserted into the generated slurm script
  - Snakemake requires one-time manual fix
- **venv creation**
  - When you want to extend an existing, very large installation
  - Venv then exists normally on disk
- **slurm**

Running, some examples

```
$ export PATH=$PWD/P/bin:$PATH
$ python3 -c "import sys;print(sys.executable)"
/scratch/project_100000002/user/CUG/P/bin/python3
$ python3 -c "import sys;print(sys.prefix)"
/LUMI_TYKKY_oX27qRR/miniconda/envs/env1
$ python3 -c 'import subprocess;subprocess.run(["srun","-
A","project_100000002","-p","debug","python3","-c","import sys;
print(sys.executable);print(sys.prefix)"])'
  srun: job 2 queued and waiting for resources
  srun: job 2 has been allocated resources
  /scratch/project_100000002/user/CUG/P/bin/python3
  /LUMI_TYKKY_oX27qRR/miniconda/envs/env1
```

Limitations

- Installation is read only, updating it requires extracting the whole squashfs image
- Aggressive path resolving breaks some things *(valid behavior if not in venv)*
  - pip installed binaries outside the container
  - some workflow managers
- Launching other containers not possible
- Tools depending on some SUID step fail
  - Host based authentication for ssh
- How safe is it to depend on the current behavior?

Future work

- Rewrite codebase in something else than Bash and Python
- Investigate options to the squashfs for more flexible updates.
- Utilize fixed image mount ordering
  - Would make filepaths appear identical on the inside and outside.
  - Fixed in apptainer and fix in progress for singularity CE
- Trash the tool in case we do enable usernamepaces and use some other tool?
  - Or if the filesystem works perfectly