Post-Processing of Large Simulation Datasets with Python Dask

HPCN Workshop 2021

Tim Horchler

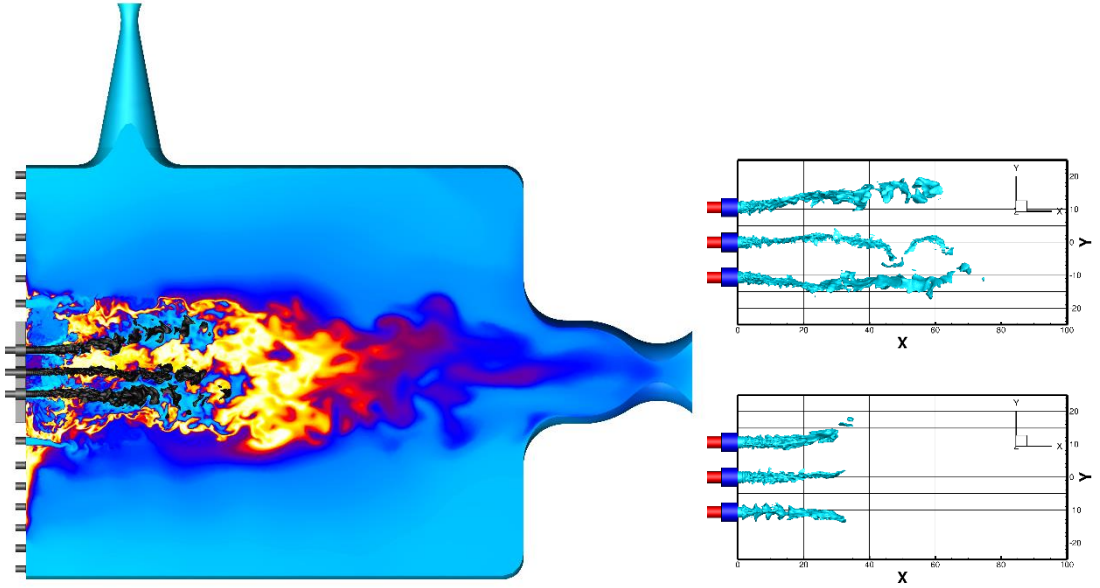
Institute of Aerodynamics and Flow Technology
Spacecraft Department
DLR Göttingen

DLR

lrz Leibniz-Rechenzentrum
der Bayerischen Akademie der Wissenschaften

HPC Produces a Lot of Simulation Data

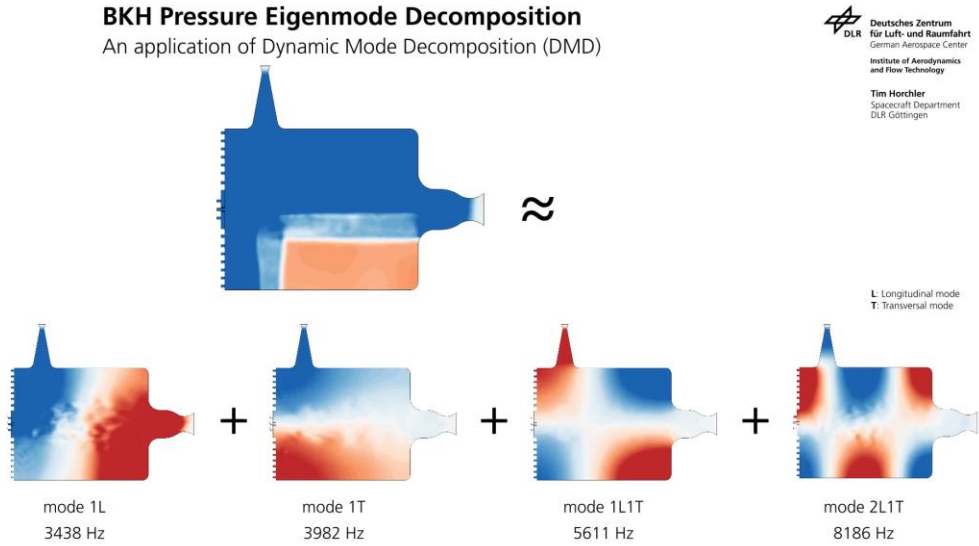
How do we process it efficiently?



Example: Combustion Chamber H

Investigating flame-acoustic interactions using numerical simulations

- Model combustion chamber from DLR Lampoldshausen
- TAU Detached-Eddy simulation (DES) on CARA and SuperMUC (project pr27ji)
- Requires real gas thermodynamics and chemical reactions
- Strong focus on the time-resolved flame response requires well sampled simulation results
- Need to analyse both surface data (small) and field data (large)

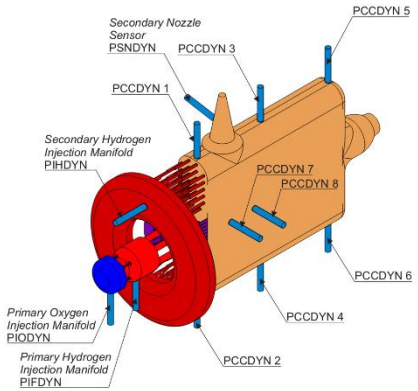


How Large is Large?

My personal definition of large data:

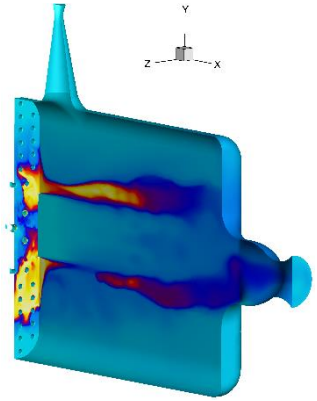
- Data is bulky, i.e. can't be moved efficiently to a different machine
- Data doesn't fit into my computer's memory
- Data can't be analyzed interactively, i.e. analysis cycle is significantly longer than a typical "prepare-brew-consume-Espresso-cycle"

Sensor data timeseries



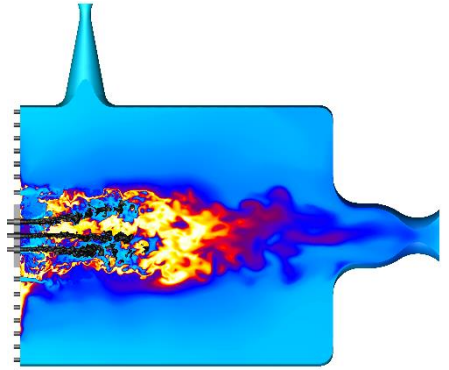
- 6 measurement points
- Total ~ 1 MB for analysis
- (Requires processing of surface data)

Surface solutions



- 216,897 surface points
- 1.5 MB per output solution variable
- Total ~ 750 MB data for analysis

Field (volumetric) solutions



- 39.5 Mio. grid points
- 300 MB per output solution variable
- Total ~150 GB data for analysis

Need 500 – 1000 samples for analysis

Parallelization in the Python Universe

***Dask** is a flexible library for parallel computing in Python.*
Dask is composed of two parts:
1. *Dynamic task scheduling*
2. *"Big Data" collections*

➢ Provide easy access to computional ressources on various parallel architectures (laptops, workstations, HPC clusters)

***xarray** (formerly `xray`) is an open source project and Python package that makes working with labelled multi-dimensional arrays simple, efficient, and fun!*

➢ Read, write and modify TAU NetCDF solution files

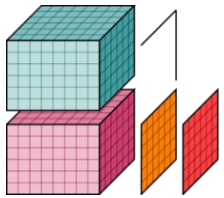
***Project Jupyter** exists to develop open-source software, open-standards, and services for interactive computing across dozens of programming languages.*

➢ Run analysis interactively and combine everything together

Parallelization in the Python Universe

Dask is a flexible library for parallel computing in Python

architectures



xarray

ackage that
efficient, and
fun!

- Read, write and modify TAU NetCDF solution files

Project Jupyter exists to develop open-source software, open-standards, and services for interactive computing across dozens of programming languages.

- Run analysis interactively and combine everything together

Disclaimer:

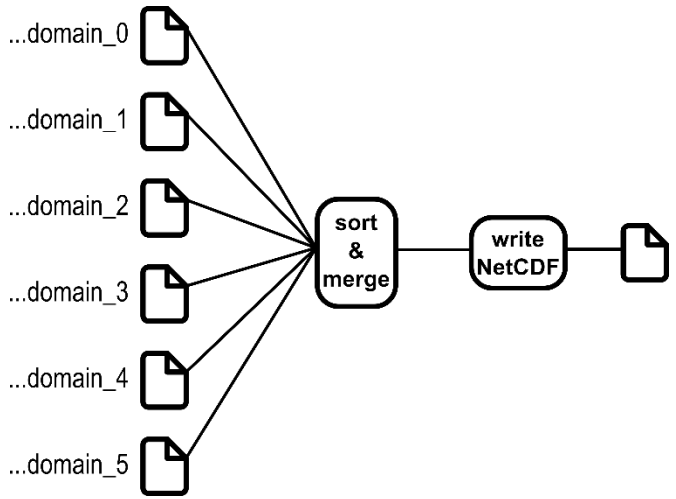
Similar functionality might exist within DLR's FlowSimulator which is not considered in this talk

Example 1: Gathering of TAU Domain Solution Files
Direct approach

- The simulation of BKH runs on 40 CARA nodes equaling 2560 CPUs
- The computational mesh is initially split up into the same number of domains
- Each domain outputs its results in a separate file

➢ **This results in not just a lot of data, but also many files**

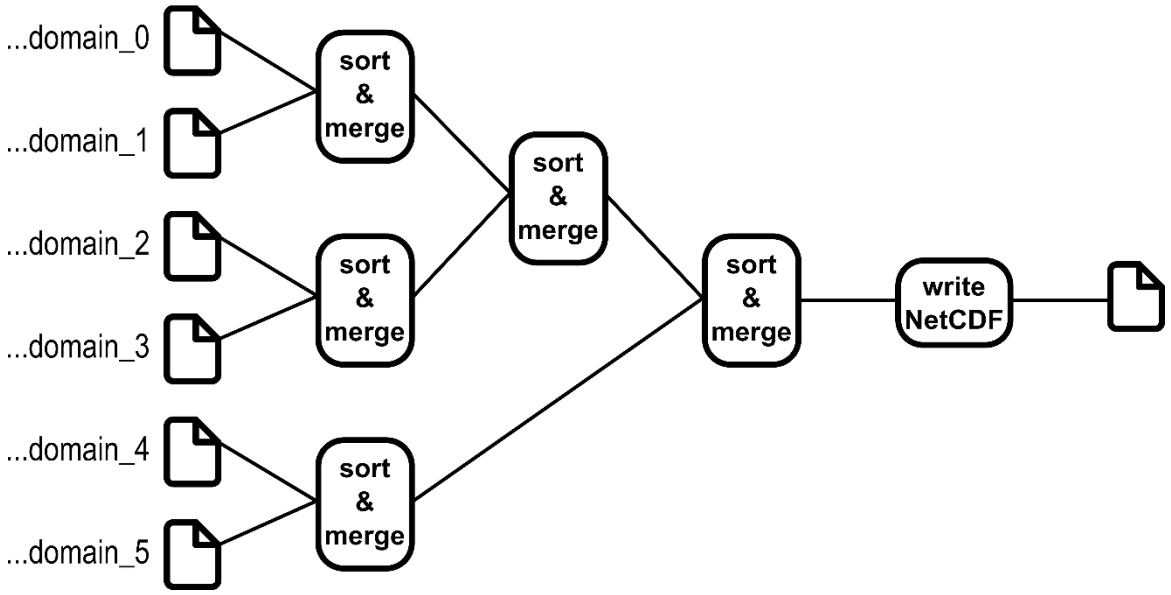
- Need to merge the domain files back into a single file for every time step (in TAU speech: `gather`)



Single-Process approach:
- All work essentially concentrated in one “sort & merge” step
- Work can’t be distributed easily

Example 1: Gathering of TAU Domain Solution Files

Tree reduction approach



Inspired by the “classic tree reduction” from:
https://examples.dask.org/delayed.html

Better idea:

- Only merge two solutions at a time
- Distribute “sort & merge” work more equally to different workers

- Still need a final “sort & merge” but only on two input solutions
- Parallel NetCDF/HDF5 writing possible but not yet realized

Approach using Python `dask.delayed`:

- Delay execution of functions
- Create execution task graph along with the dependencies
- Start executing functions once the graph is complete or the result is explicitly required

Example 1: Gathering of TAU Domain Solution Files

Gathering solutions the Python Dask way

[7]:
```
from dask.distributed import Client, progress
from dask_jobqueue import SLURMCluster
```

[8]:
```
cluster = SLURMCluster(cores=24,
                       processes=24,
                       memory="120GB",
                       project="XXXXXXX",
                       queue="naples128",
                       nanny=True,
                       walltime="00:05:00",
                       job_extra=['-J PGather',
                                  '--hint=nomultithread'])
cluster.scale(1)
```

[9]:
```
from dask.distributed import Client
client = Client(cluster)
client
```

[9]:

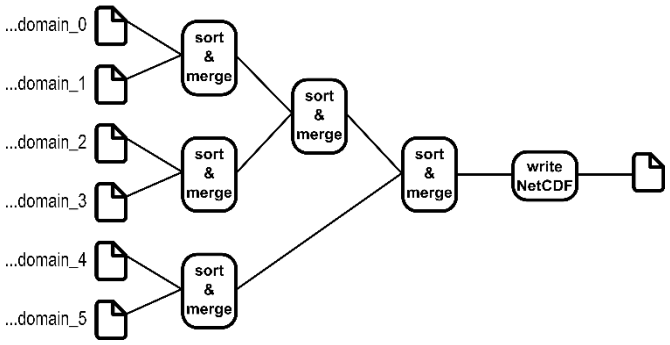
Client

Scheduler: tcp://10.247.254.1:36437
Dashboard: http://10.247.254.1:8787/status

Cluster

Workers: 0
Cores: 0
Memory: 0 B

```
domains = [client.submit(load_domain, f, keepvars) for f in domainnames]
file_base = os.path.basename(filename)

L = domains
while len(L) > 1:
    new_L = []
    carry_over = None
    if len(L) % 2:
        carry_over = L.pop()
    for i in range(0, len(L), 2):
        future = client.submit(merge_domains, L[i], L[i+1])
        new_L.append(future)
    L = new_L
    if carry_over is not None:
        L.append(carry_over)
#return write_netcdf(L[0].result(), os.path.join(outpath,file_base))
return client.submit(write_netcdf, L[0], os.path.join(outpath, file_base))
```



Example 1: Gathering of TAU Domain Solution Files

Gathering solutions the Python Dask way

```
[7]: from dask.distributed import Client, progress
     from dask_jobqueue import SLURMCluster
```

```
[8]: cluster = SLURMCluster(cores=24,
                           processes=24,
                           memory="120GB",
                           project="XXXXXXX",
                           queue="naples128",
                           nanny=True,
                           walltime="00:05:00",
                           job_extra=['-J PGather',
                                      '--hint=nomultithread'])
     cluster.scale(1)
```

```
[9]: from dask.distributed import Client
     client = Client(cluster)
     client
```

[9]:

Client

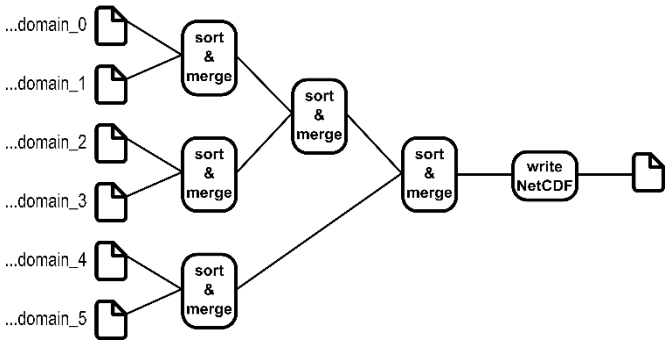
Scheduler: tcp://10.247.254.1:36427
Dashboard: http://10.247.254.1:8787/status

Cluster

Workers: 0
Cores: 0
Memory: 0 B

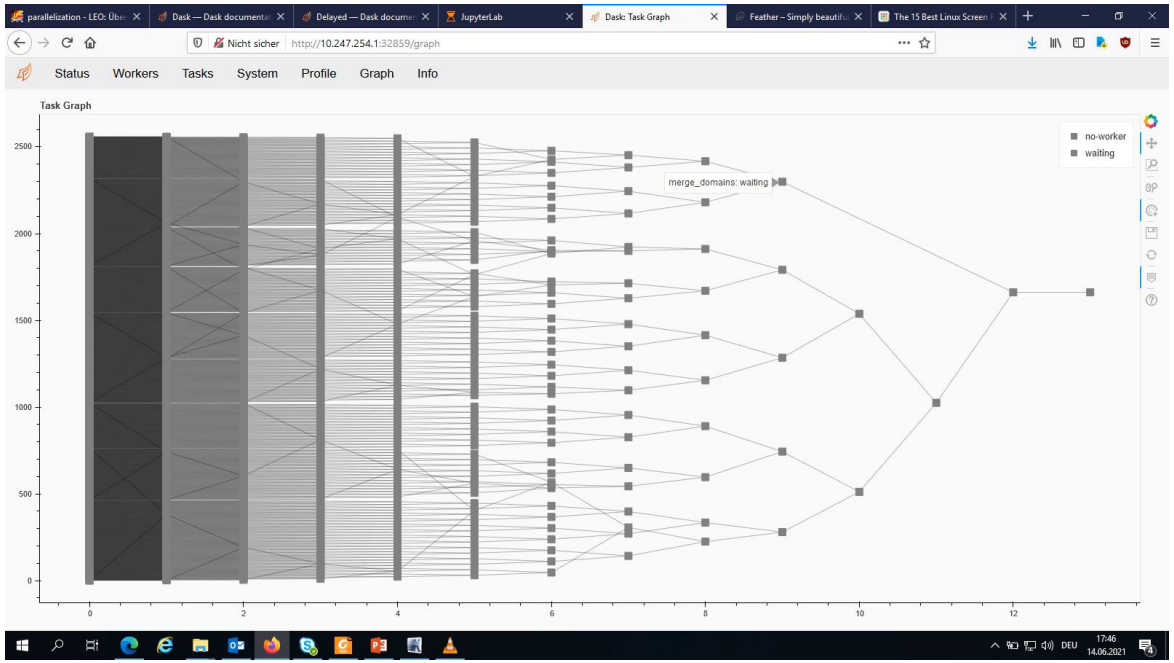
Access to dashboard

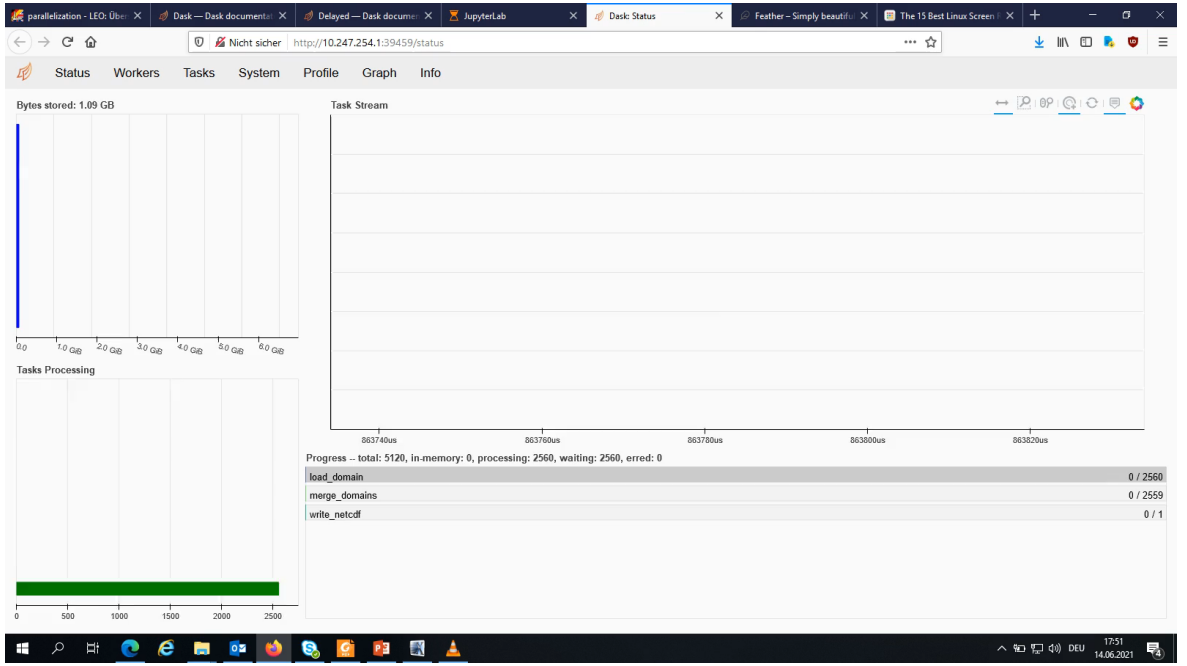
```
domains = [client.submit(load_domain, f, keepvars) for f in domainnames]
file_base = os.path.basename(filename)

L = domains
while len(L) > 1:
    new_L = []
    carry_over = None
    if len(L) % 2:
        carry_over = L.pop()
    for i in range(0, len(L), 2):
        future = client.submit(merge_domains, L[i], L[i+1])
        new_L.append(future)
    L = new_L
    if carry_over is not None:
        L.append(carry_over)
#return write_netcdf(L[0].result(), os.path.join(outpath,file_base))
return client.submit(write_netcdf, L[0], os.path.join(outpath, file_base))
```



Example 1: Gathering of TAU Domain Solution Files

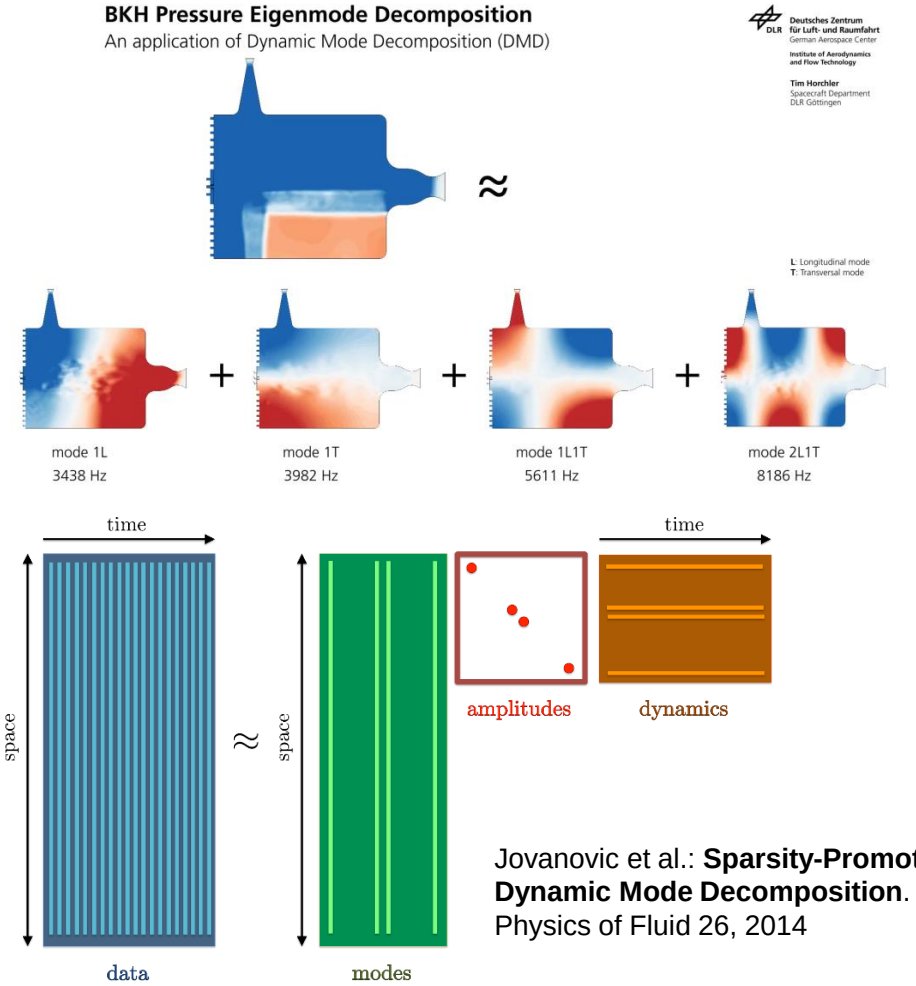
Python Dask Dashboard





Example 2: Dynamic Mode Decomposition (DMD) of Large Matrices

Current work in progress



Jovanovic et al.: **Sparsity-Promoting Dynamic Mode Decomposition**. Physics of Fluid 26, 2014

Goal: Find a low-order and single-frequency representation of the wall pressure evolution

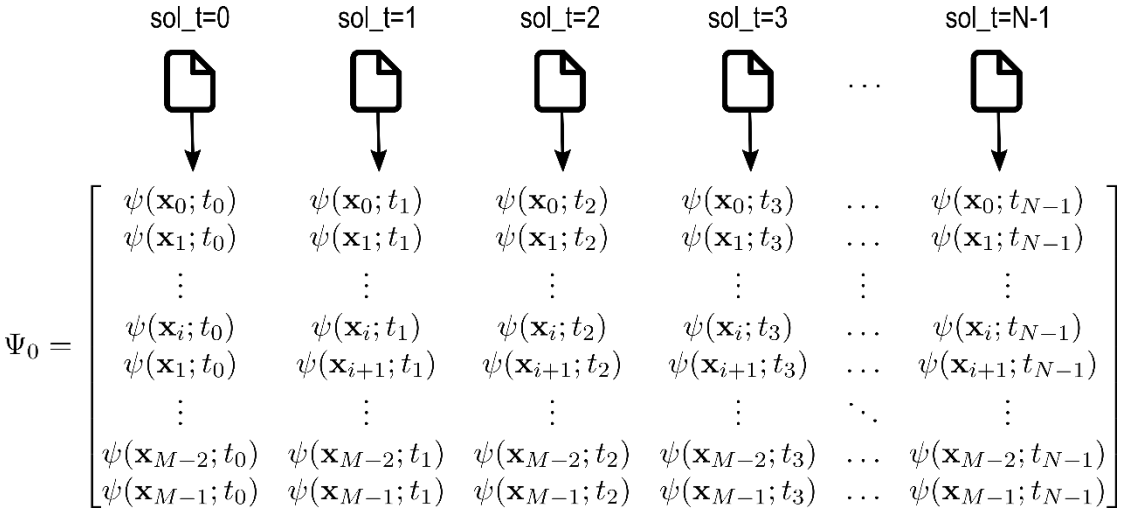
- For DMD, we need to assemble all spatial points M at all temporal snapshots N in a matrix Ψ_0 of size $\mathrm{M} \times N$; $\mathrm{M} \gg N$
- This data matrix Ψ_0 is potentially very large
- The main dimensionality reduction comes from a singular value decomposition (SVD) of Ψ_0:

$$\Psi_0 = U\,\Sigma\,V^* \text{ with}$$
$$U \in C^{M \times r}$$
$$\Sigma \in C^{r \times r}$$
$$V^* \in C^{r \times N}$$

- Here $r \leq N$ is the rank of Ψ_0

Example 2: Dynamic Mode Decomposition (DMD) of Large Matrices

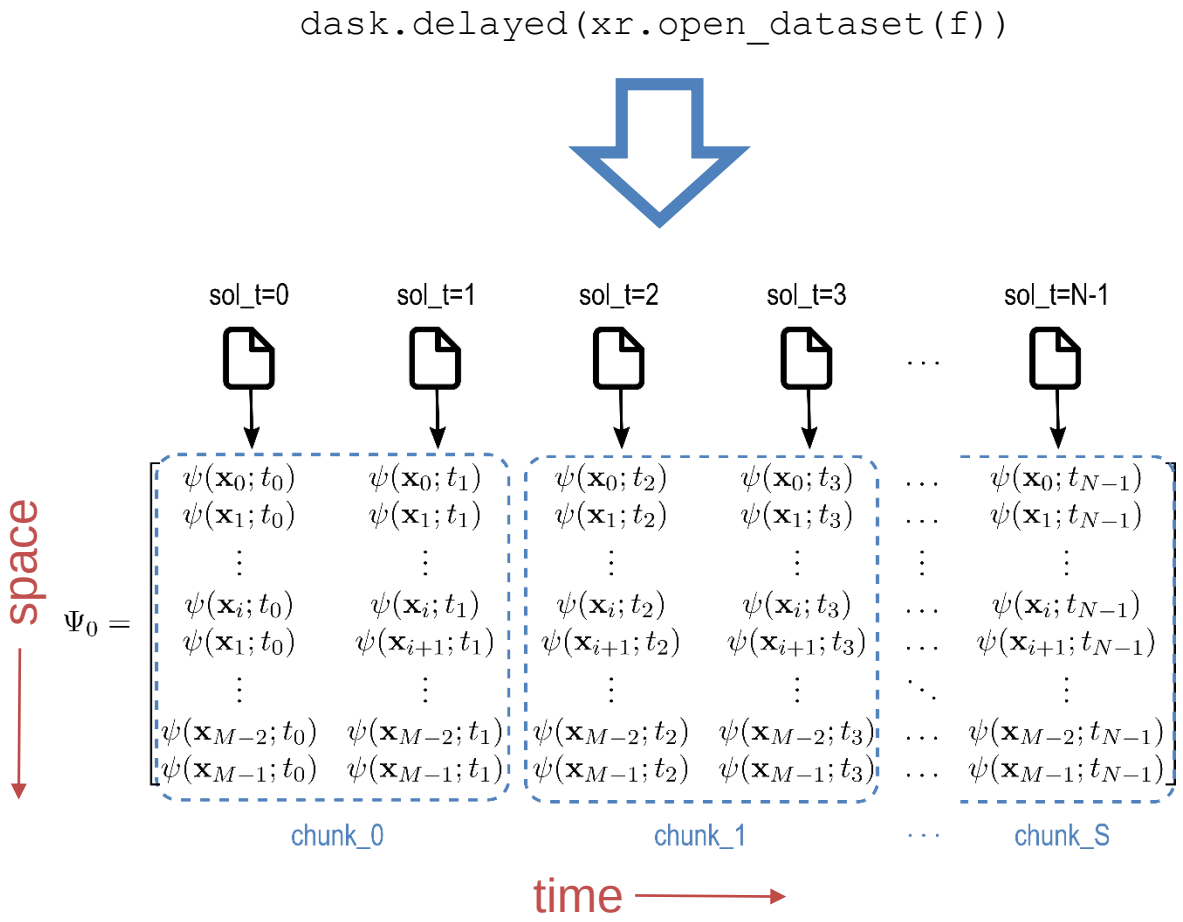
Use of Dask chunked arrays



- Dask provides many linear algebra routines `da.linalg` for operations on **chunked data**

Example 2: Dynamic Mode Decomposition (DMD) of Large Matrices

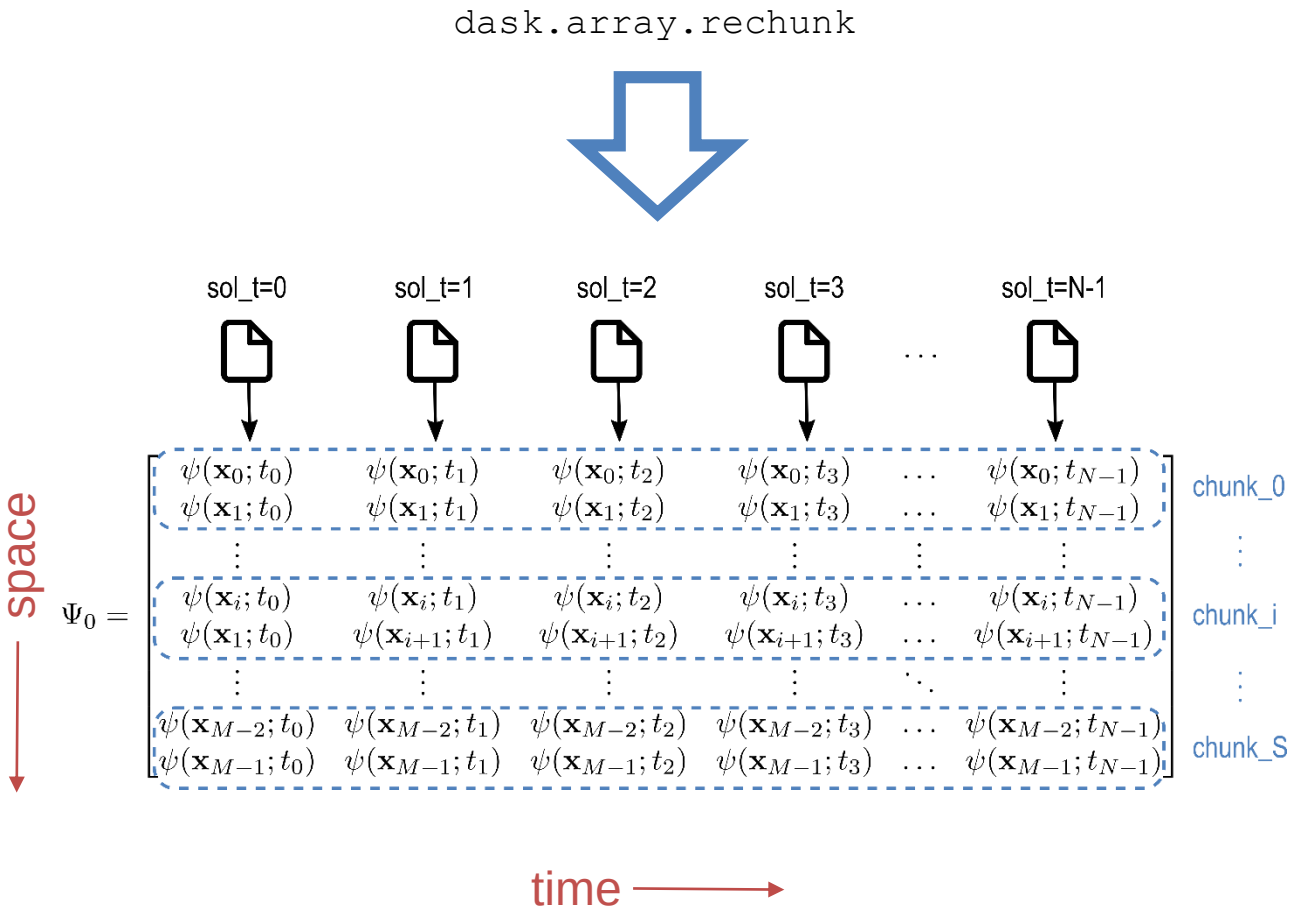
Use of Dask chunked arrays



- Dask provides many linear algebra routines `da.linalg` for operations on **chunked data**
- Use `dask.delayed` to assemble matrix in a lazy way, i.e. read data only when needed
- The natural choice for chunking is the temporal direction

Example 2: Dynamic Mode Decomposition (DMD) of Large Matrices

Use of Dask chunked arrays



- Dask provides many linear algebra routines `da.linalg` for operations on **chunked data**
- Use `dask.delayed` to assemble matrix in a lazy way, i.e. read data only when needed
- The natural choice for chunking is the temporal direction
- Currently need inefficient rechunking in space-direction: `da.linalg.svd` requires chunks in spatial direction for this application

Summary and Conclusion

- Introduced two examples from the field of computational fluid dynamics where Python can help to analyze large datasets

- This is only a small subset of what `Dask + xarray` offer, there is much more to discover, e.g.
  - Parallel image processing in combination with `scikit-image`
  - Parallel machine learning featuring `scikit-learn` and `XGBoost`

- There is a very active community around `Dask`:
  - Found many interesting examples from geophysics, meteorology and ocean dynamics
  - Found no examples from the field of CFD so far

- The next steps:
  - **Continue working on the DMD example** and try to avoid unnecessary rechunking
  - **Increase performance:** Current focus on feasibility (i.e. fit data in memory) instead of speed
  - **Automate the workflow:** Why not running the analysis automatically and let Python produce slides for me?