BIODT biodiversitydigitaltwin School & Hackaton

March 3rd - March 7th
Officine Cantelmo, Lecce IT

Introduction to High-Performance Computing

Scaling up Biodiversity Models from Laptops to Supercomputers

Tuomas Rossi, CSC – IT Center for Science

Funded by
the European Union

Outline

High-performance computing (HPC) is the use of supercomputers and computer clusters to solve advanced computation problems.

– *Wikipedia*

- Why to use supercomputers?
- What are supercomputers?
- How to utilize supercomputers?
- How are supercomputers used to scale up BioDT models?
- How to get started with using supercomputers?

BIODT
biodiversitydigitaltwin
School & Hackaton

Why to use supercomputers?

Supercomputer use cases

- Supercomputers enable solving computation problems that are impossible or too slow to solve on a standard computer
- Example modeling cases:
  - Climate
  - Material science
  - Space weather
  - Large-scale AI
  - … many more

BIODT
biodiversitydigitaltwin
School & Hackaton

What are supercomputers?

Early supercomputers

- Examples
  - CDC 6600 (1964):
    CPU 10 MHz, 982 kB RAM
  - Cray-1 (1975):
    CPU 80 MHz, 8.4 MB RAM
  - Cray X-MP (1982):
    CPU 4x ~110 MHz, 128 MB RAM
- Specialized hardware, substantially faster than other computers at the time

Cray X-MP at CSC (1989);
Image: CSC Archives

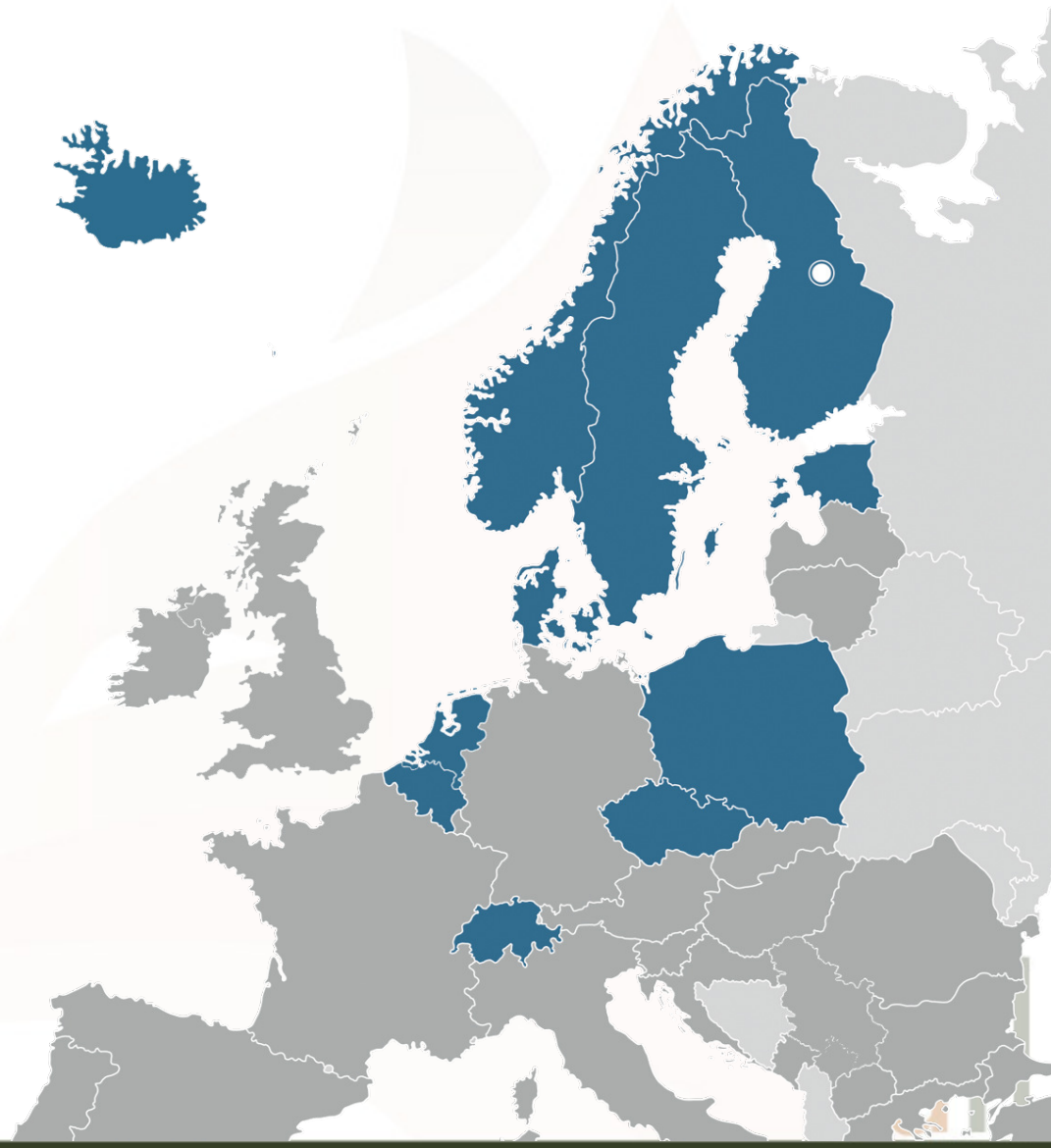
Current supercomputers

- Large computer clusters
- A lot of standard, high-end server hardware connected to each other
- High-speed interconnect between compute nodes
- Example: LUMI
  - 2064 CPU nodes, each with 128x 2.5 GHz CPUs and 256 GB – 4 TB RAM
  - 2978 GPU nodes, each with 4x MI250X GPUs and 512 GB RAM

EuroHPC JU LUMI (HPE Cray EX) at CSC (2021–)

LUMI Supercomputer

- EuroHPC pre-exascale supercomputer
  - Sustained computing power of 380 petaflop/s
  - Co-funded by EuroHPC JU and the LUMI Consortium
  - Located in Kajaani, Finland
- Green
  - 100% hydroelectric energy
  - Waste heat utilized for heating households in the city of Kajaani
- https://www.lumi-supercomputer.eu



Performance development of supercomputers

- Performance measured in Flop/s (double precision)
- Flop = floating-point operation (e.g., an addition or multiplication of two numbers)

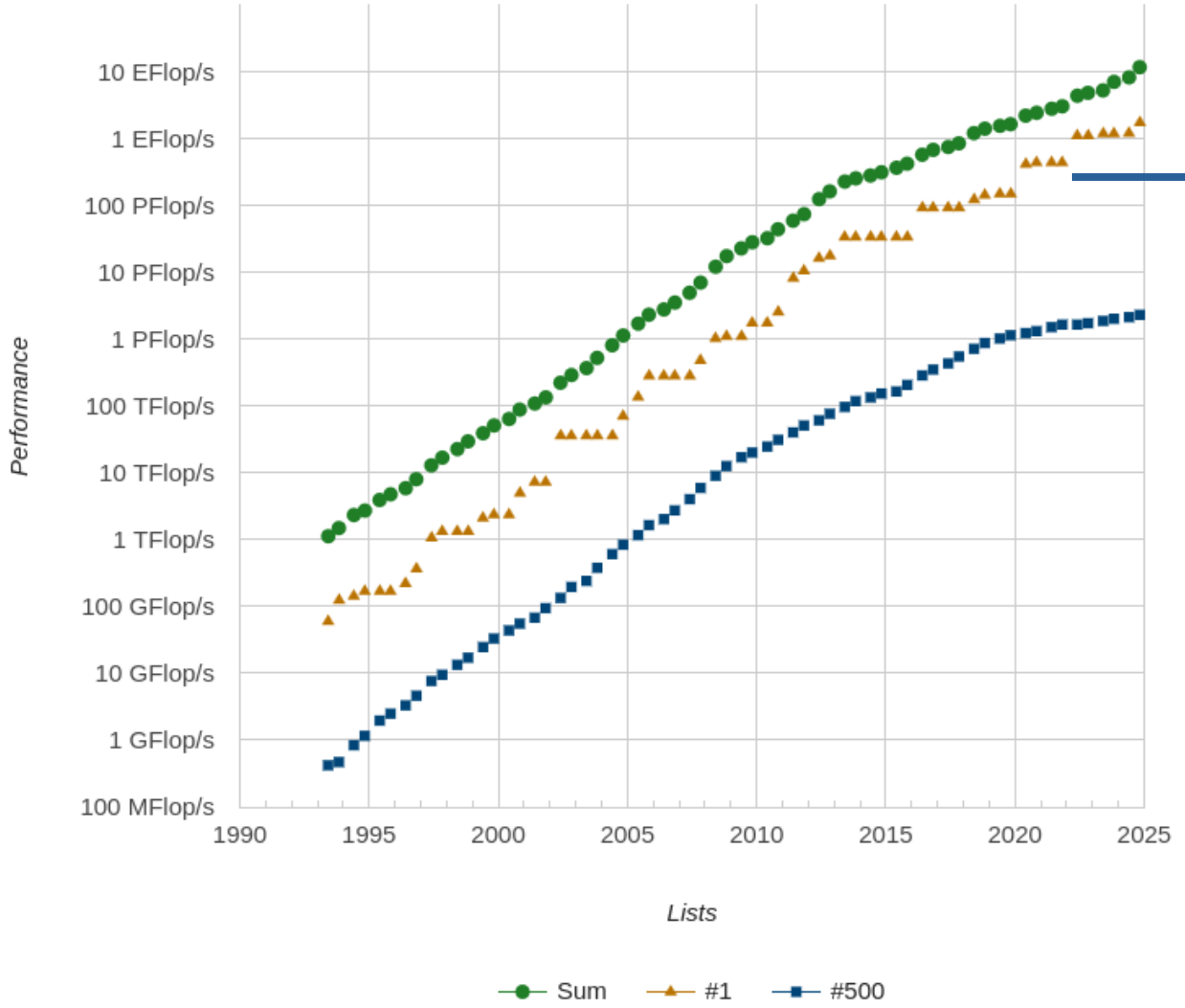


Image: top500.org

Cloud computing vs high-performance computing

- Many similarities
  - Both enable remote execution of computations on a computer system with more computing power
- Many differences
  - Different software and computing environment
  - HPC: maximal performance (incl. high-end, tuned hardware and software) → for tightly coupled workloads
  - Cloud: flexibility, fault-tolerance → for loosely coupled workloads
- HPC-as-a-Service changing the “traditional” way to use HPC
  - HEAppE https://heappe.eu/
  - LEXIS https://lexis-project.eu

Illustration of computer clusters

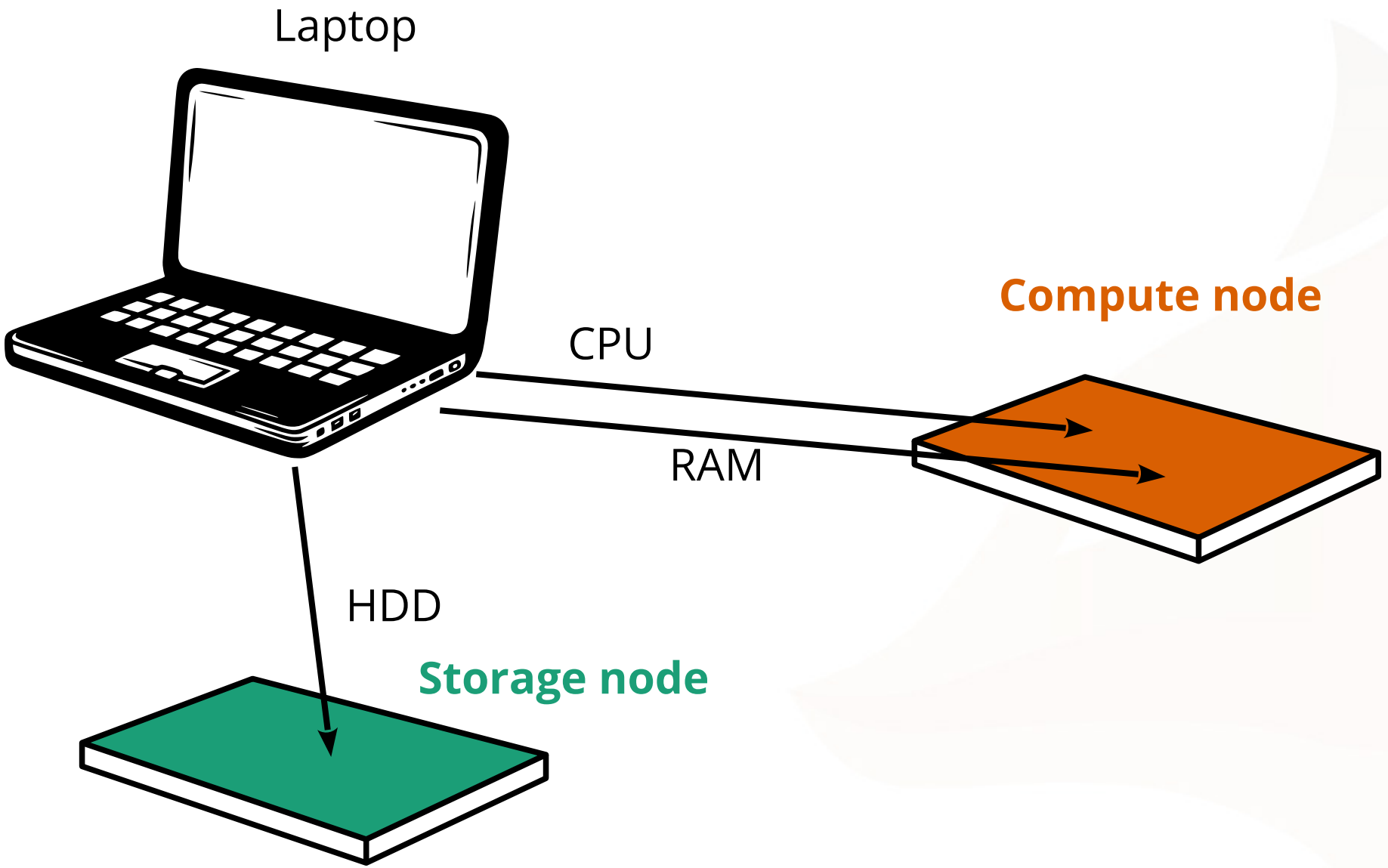


Illustration of computer clusters

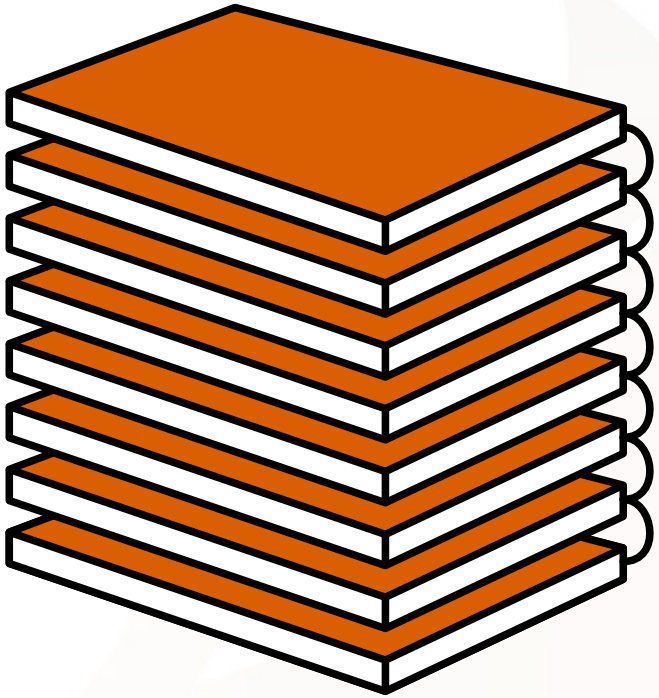


Illustration of computer clusters

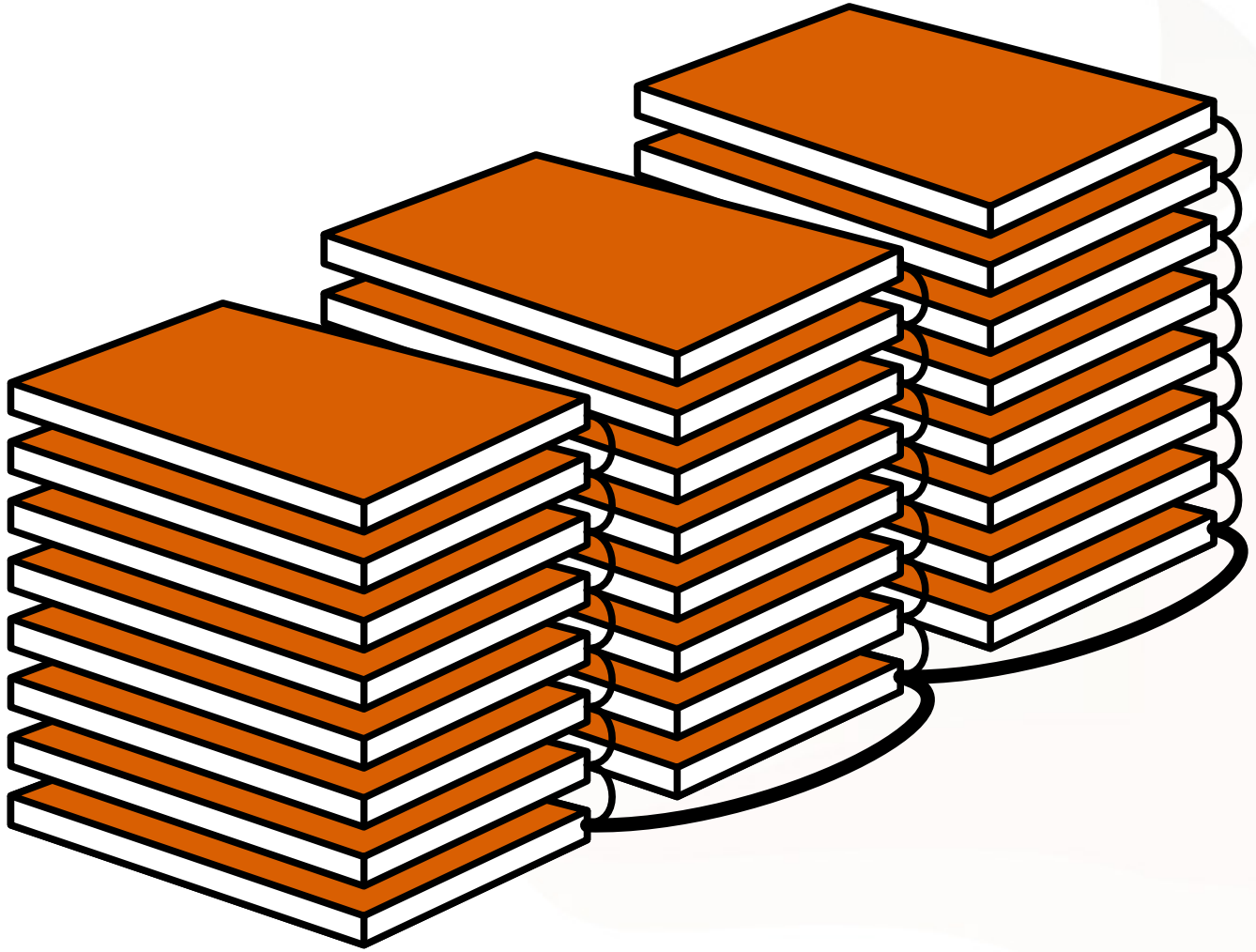


Illustration of computer clusters

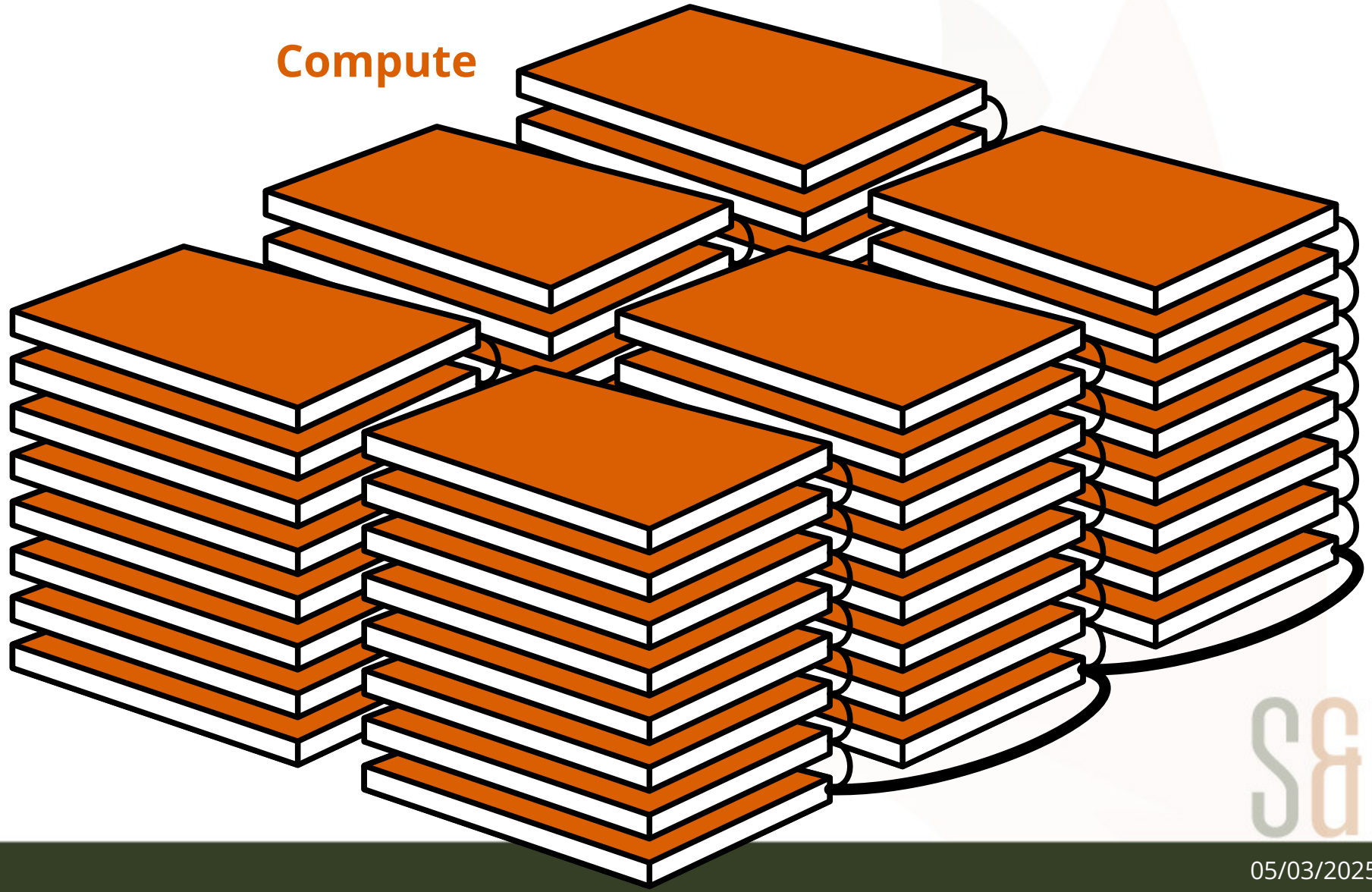
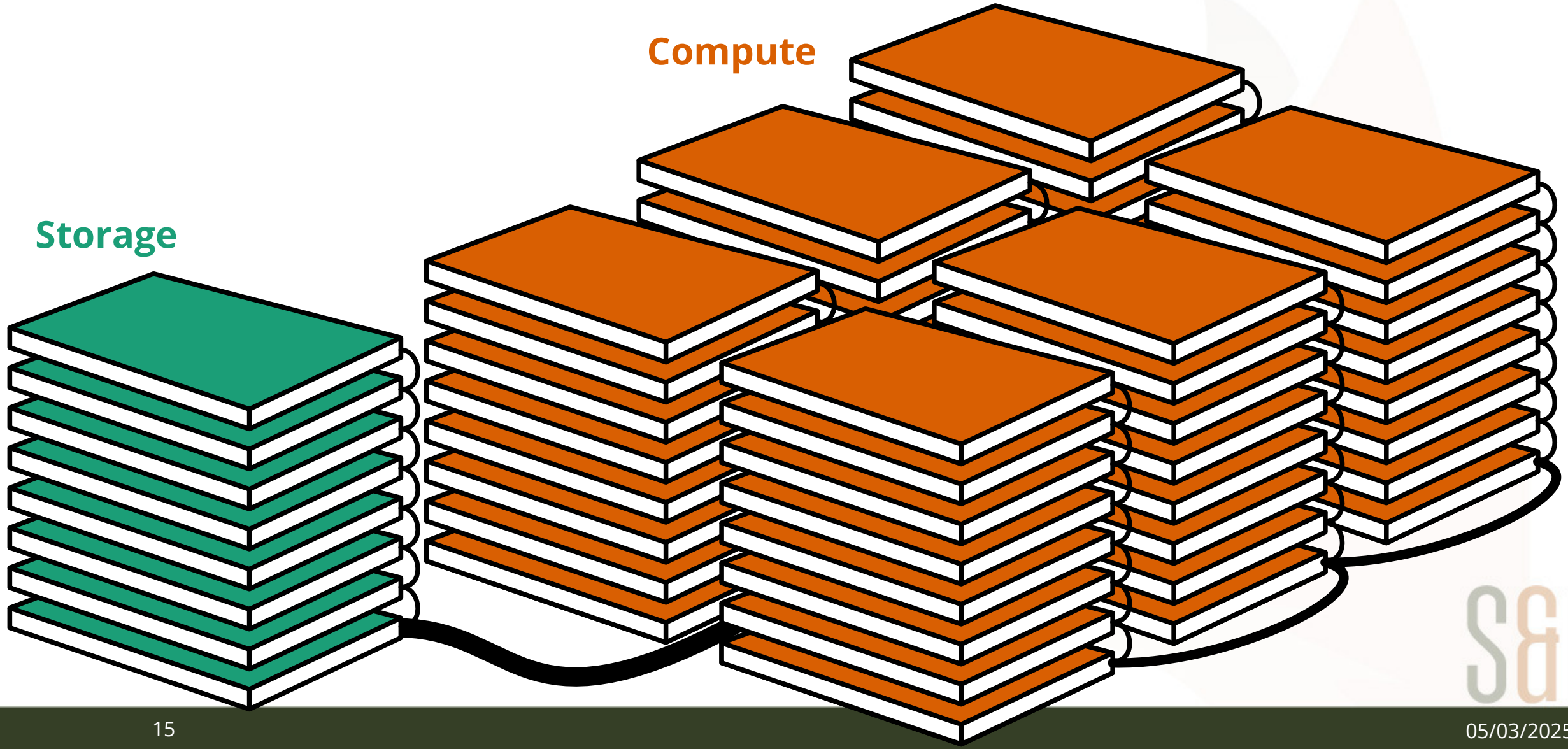
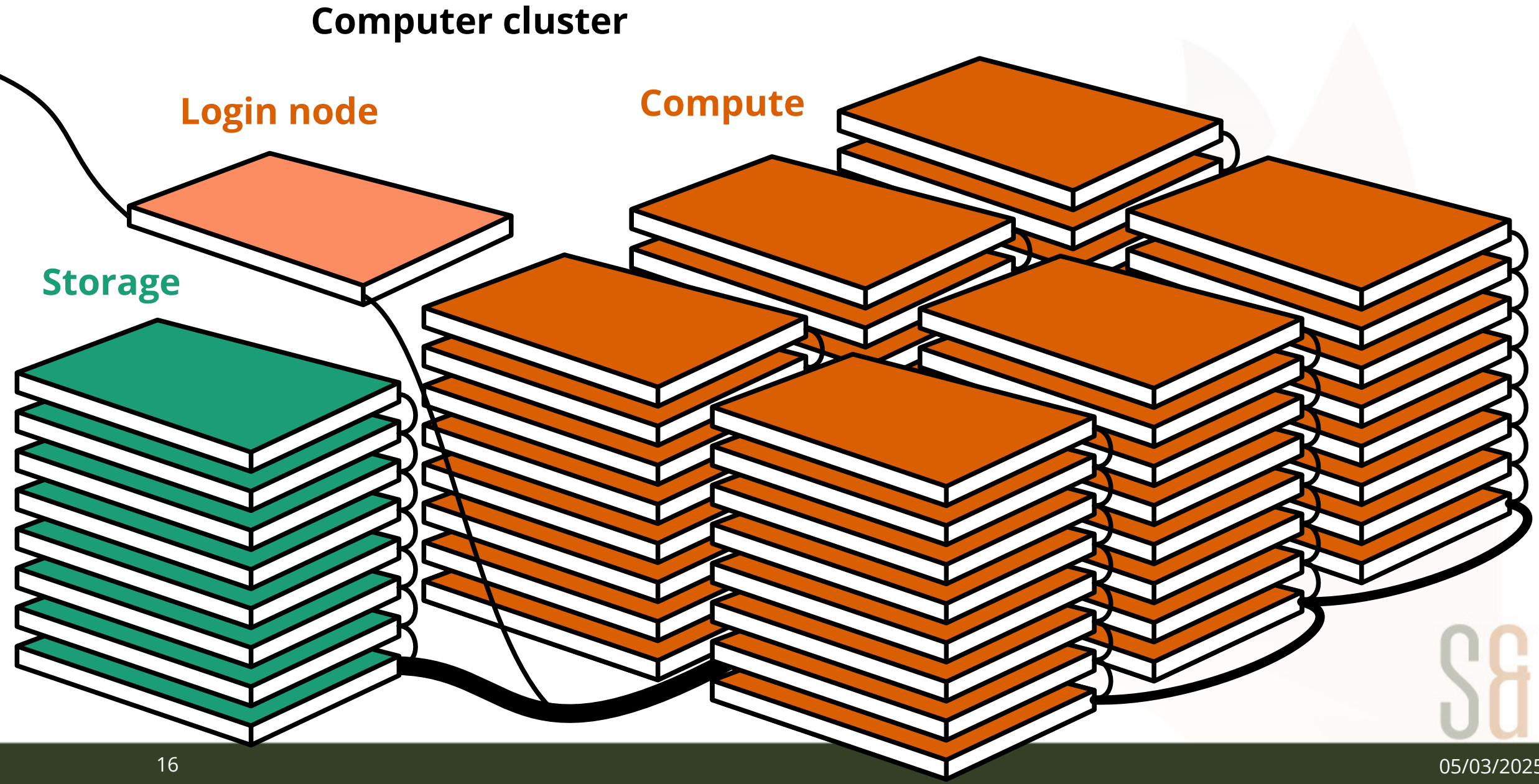


Illustration of computer clusters




Computer cluster
Login node
Compute
Storage

Computer clusters and supercomputers

- A computer cluster is a set of computers connected to each other
- The cluster is visible for the user as a single system
- Most current supercomputers are very large computer clusters
  - Current trend in HPC: most of the computing power is in graphics processing units (GPUs) instead of central processing units (CPUs)
    → More on this later

A supercomputer is not a single “super fast” processor, but a set of hundreds of thousands (or more) processing units that *can be programmed* to work together

→ *Parallel algorithms* needed to utilize supercomputer

BIODT
biodiversitydigitaltwin
School & Hackaton

How to utilize supercomputers?

Parallel algorithms - Example: Computing a sum of numbers

```
23 + 99 = ...
```

Parallel algorithms - Example: Computing a sum of numbers

```
23 + 99 + 97 + 62 = ...
```

Parallel algorithms - Example: Computing a sum of numbers

```
  23 + 99 + 97 + 62 + 40 + 30 + 72 + 19 + 88 + 12
= ...
```

Parallel algorithms - Example: Computing a sum of numbers

```
  23 + 99 + 97 + 62 + 40 + 30 + 72 + 19 + 88 + 12
+ 14 + 66 +  4 + 61 + 49 + 58 + 39 + 28 + 86 + 84
+ 65 + 92 + 49 + 48 + 93 + 75 + 32 + 82 + 92 + 75
+ 31 +  8 + 55 + 70 +  1 + 80 + 23 + 78 + 73 + 62
+ 11 + 31 + 99 + 50 + 26 + 82 + 98 + 22 + 82 + 48
+ 85 + 69 + 71 + 60 + 27 + 55 + 29 +  7 +  9 + 99
+ 86 + 36 + 95 + 50 + 94 + 87 + 69 +  7 + 59 + 85
+ 22 + 50 +  5 + 70 +  5 + 59 + 94 + 69 + 48 + 50
+ 45 + 73 +  2 + 64 + 93 + 50 + 72 +  5 + 66 + 21
+ 84 + 33 + 12 + 58 + 35 + 42 + 63 + 33 +  5 + 22
+ 70 + 91 + 71 + 97 + 79 + 13 +  2 +  8 +  3 + 41
+ 50 + 74 + 28 + 87 + 39 + 41 +  2 + 72 + 23 + 19
= ...
```

Parallel algorithms - Example: Computing a sum of numbers

```
   23 + 99 + 97 + 62 + 40 + 30 + 72 + 19 + 88 + 12    1
+ 14 + 66 +  4 + 61 + 49 + 58 + 39 + 28 + 86 + 84    2
+ 65 + 92 + 49 + 48 + 93 + 75 + 32 + 82 + 92 + 75    3
+ 31 +  8 + 55 + 70 +  1 + 80 + 23 + 78 + 73 + 62    4
+ 11 + 31 + 99 + 50 + 26 + 82 + 98 + 22 + 82 + 48    5
+ 85 + 69 + 71 + 60 + 27 + 55 + 29 +  7 +  9 + 99    6
+ 86 + 36 + 95 + 50 + 94 + 87 + 69 +  7 + 59 + 85    7
+ 22 + 50 +  5 + 70 +  5 + 59 + 94 + 69 + 48 + 50    8
+ 45 + 73 +  2 + 64 + 93 + 50 + 72 +  5 + 66 + 21    9
+ 84 + 33 + 12 + 58 + 35 + 42 + 63 + 33 +  5 + 22   10
+ 70 + 91 + 71 + 97 + 79 + 13 +  2 +  8 +  3 + 41   11
+ 50 + 74 + 28 + 87 + 39 + 41 +  2 + 72 + 23 + 19   12
= ...
```

Parallel algorithms - Example: Computing a sum of numbers

```
  542
+ 489
+ 703
+ 481
+ 549
+ 511
+ 668
+ 472
+ 491
+ 387
+ 475
+ 435
= ...
```

```
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
```

Parallel algorithms - Example: Computing a sum of numbers

```
   542 + 489      1
                  _
+ 703 + 481      3
                  _
+ 549 + 511      5
                  _
+ 668 + 472      7
                  _
+ 491 + 387      9
                  _
+ 475 + 435     11
                  _
= ...
```

Parallel algorithms - Example: Computing a sum of numbers

```
   1031          1
                 ___
+  1184          3
                 ___
+  1060          5
                 ___
+  1140          7
                 ___
+  878           9
                 ___
+  910          11
                 ___
=  ...
```

Parallel algorithms - Example: Computing a sum of numbers

```
   1031 + 1184   1
                 ____
                 ____
                 ____
                 ____
+ 1060 + 1140    5
                 ____
                 ____
                 ____
+ 878 + 910      9
                 ____
                 ____
                 ____
= ...
```

Parallel algorithms - Example: Computing a sum of numbers

```
   2251          1
                 _
                 _
                 _
                 _
+  2200          5
                 _
                 _
                 _
+  1788          9
                 _
                 _
                 _
=  ...
```

Parallel algorithms - Example: Computing a sum of numbers

```
2251 + 2200 + 1788      1
                      ____
                      ____
                      ____
                      ____
                      ____
                      ____
                      ____
                      ____
                      ____
                      ____
                      ____
                      ____
                      ____
                      ____

= ...
```

Parallel algorithms - Example: Computing a sum of numbers

= **6239**

1

Parallel algorithms - Example: Computing a sum of numbers - Observations

- A sum of numbers can be calculated efficiently in parallel
- Parallel computation causes *overhead* (extra time cost due to parallelization)
  - Distribution of the initial data (numbers) to individual computers
  - Gathering partial sums to individual computers to perform final summation
  - Managing and coordinating individual sub-tasks
- Very small tasks should not be parallelized because the overhead can outweigh the benefits of faster computation!

How much speedup can you get by parallelization?

- Amdahl's law: The speedup is limited by the fraction of time spent by the algorithm to be parallelized / optimized
- Example workflow:
  1) Preprocessing: 2 minutes
  2) Heavy computation: 17 minutes
  3) Postprocessing: 1 minute
- Assuming ideal parallelization for the heavy computation
  - In reality, overhead limits the benefits of the parallelization even further

Number of processors	Ideal execution time (minutes)	Parallel efficiency
1	20.0	100%
2	11.5	87%
4	7.3	67%
8	5.1	49%
16	4.1	31%
32	3.5	18%
64	3.3	10%

Speedup is not everything!

- Choose the number of processors based on efficient resource utilization – not only by the execution time
  - Wasted resources consume electricity and your project's resource quota
- Resource usage is measured in processor time
- Using a lot of extra resources for a minimal gain in execution time is not usually worth it

Number of processors	Ideal execution time (minutes)	Resource cost (processor-min)
1	20.0	20.0
2	11.5	23.0
4	7.3	29.2
8	5.1	40.1
16	4.1	65.6
32	3.5	112.0
64	3.3	211.2

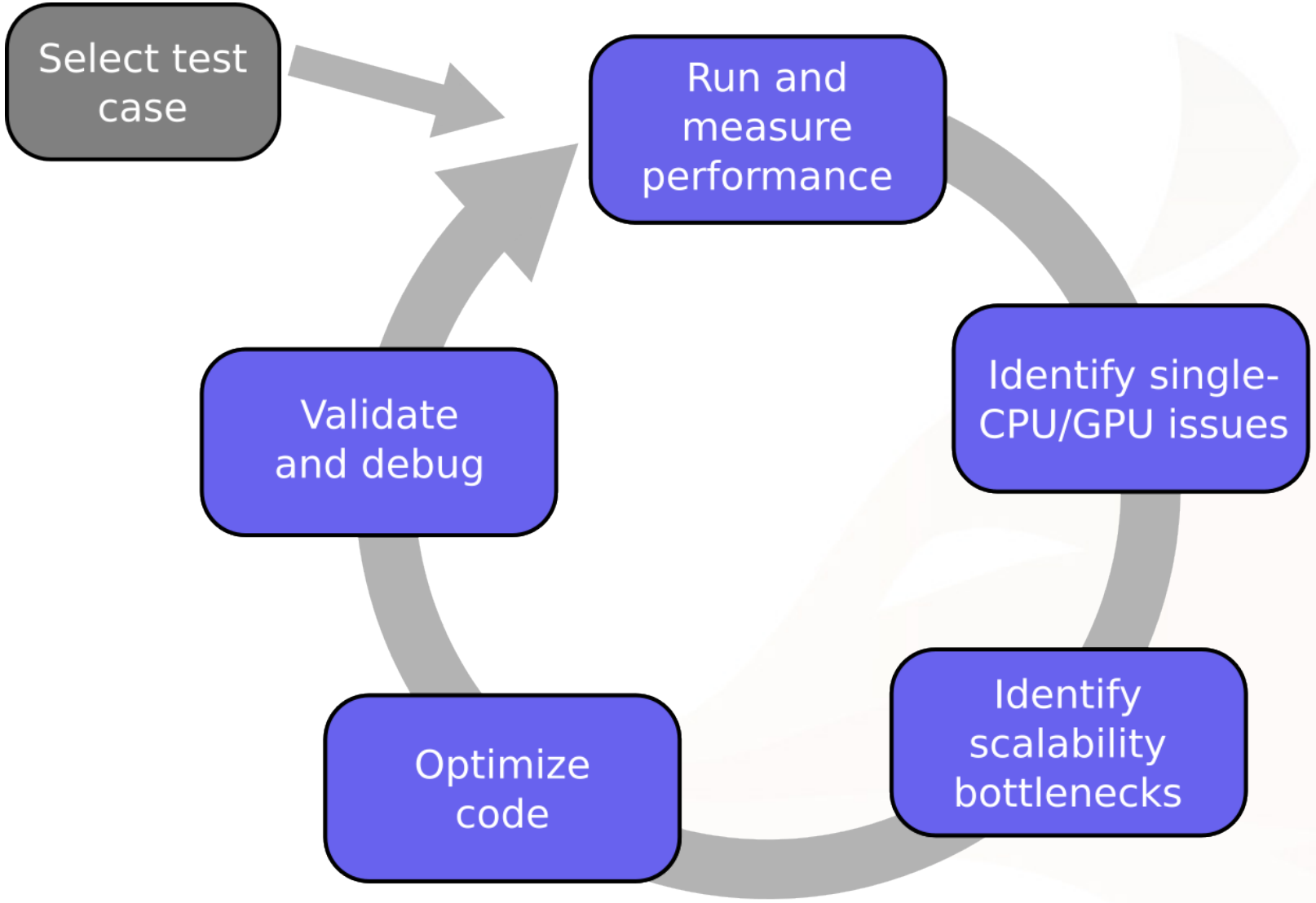
- Parallel programming can be performed in many levels
  - Managing individual processes and threads *within the application*: Message-Passing Interface (MPI), OpenMP, HIP/CUDA, …
  - Utilizing parallelized libraries *within the application*: NumPy, Pytorch, TensorFlow, …
  - Parallelizing the workflow *without modifying the application*: Scripting, GNU Parallel, HyperQueue, …

- Many BioDT use cases have been parallelized at the workflow level

BIODT
biodiversitydigitaltwin
School & Hackaton

Performance optimization

High-performance is not all about parallelization



Example: Use the right libraries and use them in a right way

```
import numpy as np
np.random.seed(123)

# Prepare data
N = 1024
a = np.random.rand(N, N)
b = np.random.rand(N, N)
c = np.zeros([N, N])

# Heavy computation; not using library calls
for i in range(N):
    for j in range(N):
        for k in range(N):
            c[i, j] += a[i, k] * b[k, j]
        if i == j:
            c[i, j] += i * j

print(c)
```

Takes 366 seconds!

Example: Use the right libraries and use them in a right way

```
import numpy as np
np.random.seed(123)

# Prepare data
N = 1024
a = np.random.rand(N, N)
b = np.random.rand(N, N)
c = np.zeros([N, N])

# “Heavy” computation; use library calls
d = np.matmul(a, b) + np.diag(np.arange(N)**2)

print(c)
```

Takes 0.05 seconds!

Results are the same!

Code optimization

- Optimizing the performance of larger codes is not usually easy
  - Start with profiling (measuring performance) to detect the computational bottleneck

- Example from BioDT: Cultural Ecosystem Services pDT
  - See the separate presentation

BIODT
biodiversitydigitaltwin
School & Hackaton

General-purpose graphics processing units (GPUs)

Extreme parallelization: GPUs

- Current trend in HPC: most of the computing power is provided by general purpose graphics processing units (GPUs)
  - LUMI:
    - GPU performance: 380 PFlop/s (power: 7.1 MW) → 54 GFlop/J
    - CPU performance: 6.3 PFlop/s (power: 1.2 MW) → 5.3 GFlop/J
- GPUs are more energy efficient for heavy number crunching
- GPUs are very efficient for executing the same computation over a large amount of data
  - For example, matrix-matrix multiplication
  - Not efficient if there is not enough parallelism → use CPU instead

Programming GPUs

- GPUs are accelerator devices
  - CPU runs the main code and offloads computation tasks to GPUs
- GPUs utilize their own programming languages and frameworks
  - Extensive reprogramming often needed to port CPU code to GPUs
- Many GPU-accelerated libraries exist
  - Makes it easier for accelerating CPU code with GPUs

- Example from BioDT: Hmsc-HPC code used in different pDTs
  - https://github.com/hmsc-r/hmsc-hpc

CSC Summer School in High-Performance Computing

Dates: June 24 - July 3, 2025

Location: Nuuksio, Espoo, Finland

Highlights:

- Learn programming (CPUs and GPUs) at supercomputer scale
- Training sessions with CSC experts
- Networking with peers and professionals

https://csc.fi/en/training-calendar/csc-summer-school-in-high-performance-computing-2025/

BIODT
biodiversitydigitaltwin
School & Hackaton

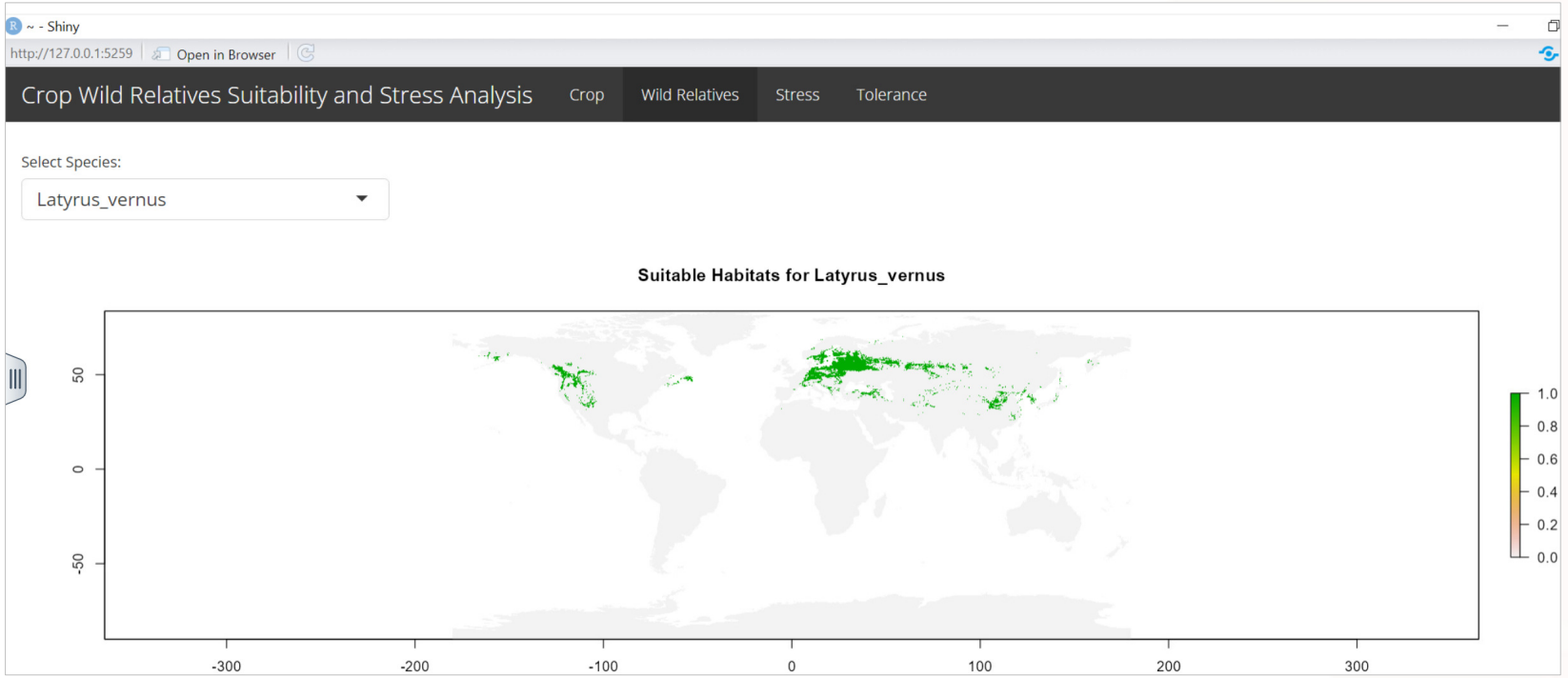
How are supercomputers used to scale up BioDT models?

Example: Crop wild relatives (CWR) pDT

- Computation workflow in CWR pDT:
  1) Download input data:
     - Species occurrence data from e.g. GBIF
     - Climate and soil data from e.g. Copernicus
     - Light computation, mainly waiting for data downloads
  2) Compute habitat suitability maps for every species (ensemble SDM)
     - Heavy computation, independent for each species
  3) Visualize results
     - Light computation, interactive

Example: Crop wild relatives (CWR) pDT

- Let's calculate habitat suitability map for spring pea (Lathyrus vernus):
  1) Download GBIF records (whole Lathyrus): queue time ~10 mins
  2) Compute habitat suitability map: ~45 mins (on a single CPU core)
- It is not a problem to execute this analysis on a laptop!



Example: Crop wild relatives (CWR) pDT

- Let's calculate the whole Lathyrus genus (peavines) with enough data:

Lathyrus pratensis	Lathyrus linifolius	Lathyrus niger
Lathyrus sylvestris	Lathyrus tuberosus	Lathyrus heterophyllus
Lathyrus vernus	Lathyrus latifolius	Lathyrus palustris
Lathyrus japonicus	Lathyrus odoratus	Lathyrus hirsutus
Lathyrus oleraceus	Lathyrus pannonicus	Lathyrus pisiformis
Lathyrus sphaericus	Lathyrus nissolia	Lathyrus aphaca
Lathyrus laevigatus	Lathyrus venetus	Lathyrus grandiflorus
Lathyrus filiformis	Lathyrus cicera	Lathyrus angulatus
Lathyrus vestitus	Lathyrus clymenum	Lathyrus annuus
Lathyrus tingitanus	Lathyrus setifolius	Lathyrus ochroleucus
Lathyrus gmelinii	Lathyrus ochrus	Lathyrus sativus
Lathyrus saxatilis	Lathyrus humilis	Lathyrus tremolsianus

  - 36 species
- Computing one-by-one on a laptop: ~27 hours
  - Still doable

Example: Crop wild relatives (CWR) pDT

- Let's calculate wild relatives of more crops (potatoes, rices, …):

Solanum	Ensete	Musa	Oryza	Triticum
Zea	Sorghum	Hordeum	Glycine	Manihot
Ipomoea	Colocasia	Dioscorea	Arachis	Pennisetum
Cicer	Lens	Phaseolus	Vigna	Saccharum

  - 20 genera; if ~30 species per genus: ~600 species
- Computing one-by-one on a laptop: ~19 days
  - A bit too long?

Example: Crop wild relatives (CWR) pDT

- Computations for each species are independent of each other
  → Possible to process in parallel with minimal overhead
- Let's use LUMI and take 5 CPU nodes (640 CPU cores in total):
  - Compute all 600 species in parallel: ~45 mins
    (same as for a single species)

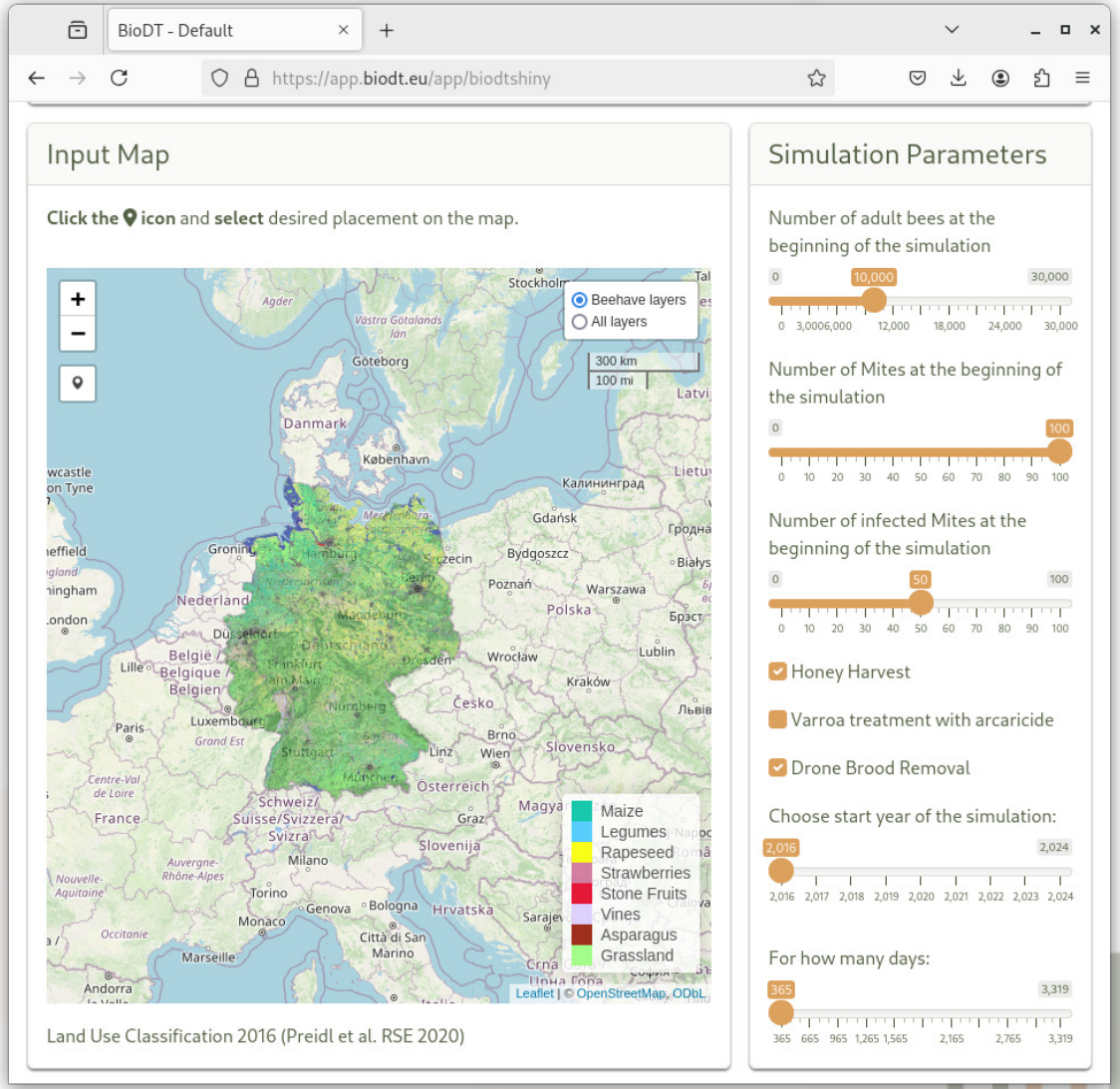
- Further speedup would be possible by parallelizing the execution of ensemble SDM (e.g., across different SDM methods and model replications)

Example: Crop wild relatives (CWR) pDT

- Other reasons for using HPC:
  - Each species computation requires ~32 GB RAM peak
    - Run 4 in parallel: ~128 GB RAM needed
    - Run 128 in parallel: ~4096 GB RAM needed
  - Each species computation produces ~5 GB data
    - Lathyrus (36 species): ~180 GB data
    - More crops (~600 species): ~3000 GB data

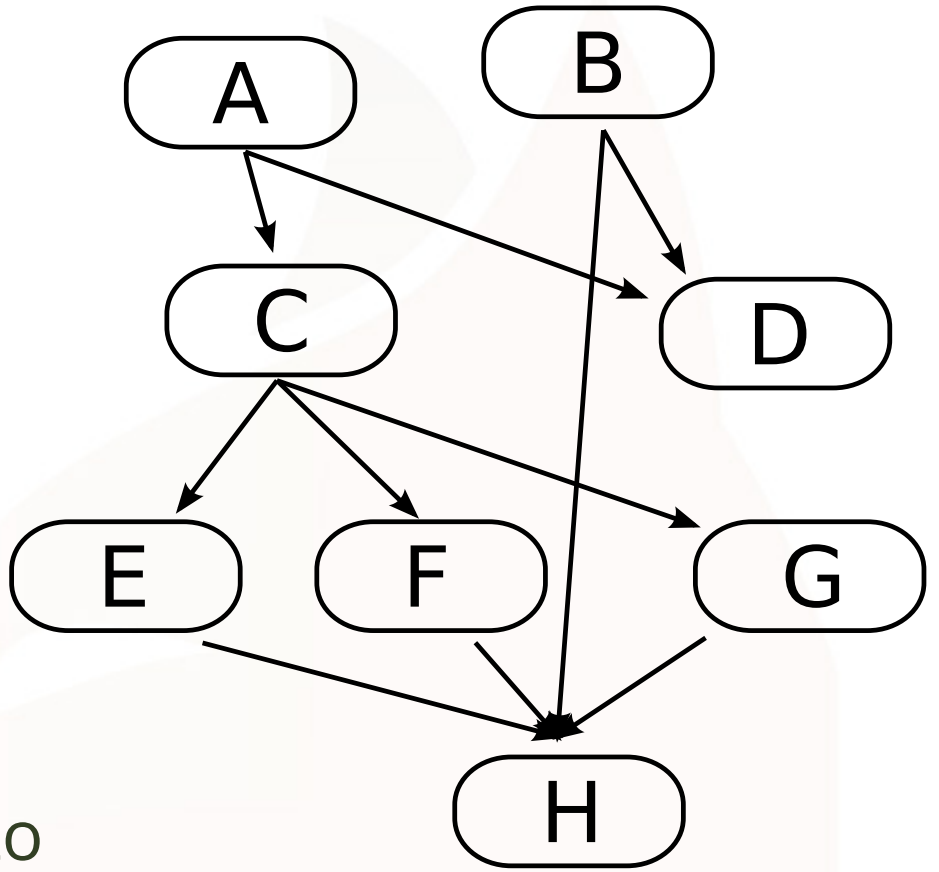
Another example: Honeybee pDT

- Computation workflow in Honeybee pDT:
  1) Download input data
  2) Compute honey production at a given coordinate and with the defined parameters
     - Independent computation for each location / set of parameters → Possible to process in parallel
  3) Visualize



- CWR and many other BioDT use cases utilize complex R and/or Python environments
  - Environment packaged using *containers*
- Processing large amounts of independent input data
  - Execution parallelized using tools such as *hyperqueue*

- HyperQueue is a workflow executor
  - https://it4innovations.github.io/hyperqueue/
- User defines
  - Computation tasks and their dependencies (task graph)
  - Amount of resources (e.g., the number of available CPU cores)
- HyperQueue does
  - Executes the tasks in parallel according to the order encoded by the dependencies
- Used in CWR to process species in parallel



BIODT
biodiversitydigitaltwin
School & Hackaton

Containers

- Traditionally, the application source code is compiled using the supercomputer's optimized compilers and libraries
  - Enables maximal performance and tuning
  - Drawback: Programs need to be recompiled for every target machine
- Alternatively, the application can be executed in a *container*
  - Encapsulates the application software and all of the required software environment
  - Portable: Can be prepared anywhere and executed anywhere
  - Drawback: Machine-specific performance tuning not possible if aiming to keep container portable

Container terminology

- *Container recipe*
  - A list of commands defining the contents of the container
  - Examples:
    - `pip install my_python_packages`
    - `install.packages(my_r_packages)`
- *Container image*
  - Built by executing the container recipe
  - Fixed, versionable environment
    - Even if the recipe would not define package versions, the container image has versions frozen to the moment when the image was built

- Useful for encapsulating complex software environments
  - R and/or Python environments with hundreds of packages
  - Other required software: GDAL, Java, …
- Versioned images distributed via GitHub package registry
  - https://github.com/orgs/BioDT/packages
  - Image built once, then reused on a laptop, HPC, cloud, …

BIODT
biodiversitydigitaltwin
School & Hackaton

Using LUMI supercomputer

- Every user needs
  - A personal account
  - Membership in a computational project with resources
    - CPU hours
    - GPU hours
    - Storage hours

Submitting computation jobs

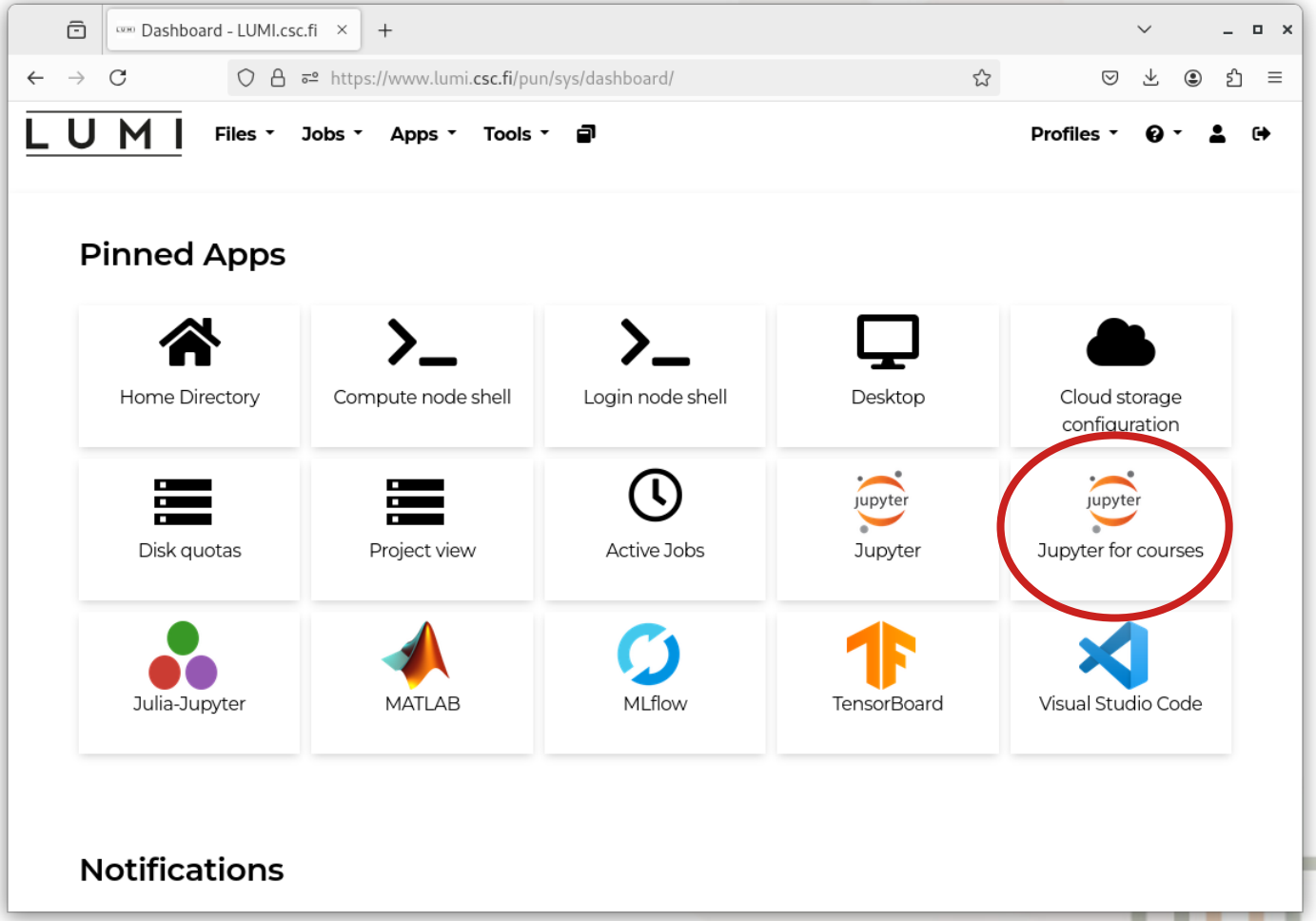
- Supercomputer is a shared resources and computations are submitted to a batch queuing system
  - Each job defines the maximum run time and the amount of reserved resources (CPUs, GPUs, RAM, …)
  - The usage is billed by the reserved resources and the actual run time
  - The job is killed when the maximum run time is reached
- Queuing system ensures efficient utilization of resources and fair use
- See examples in Jupyter notebooks

Logging in to LUMI

- Logging in
  - Using ssh client
  - Web interface https://www.lumi.csc.fi/

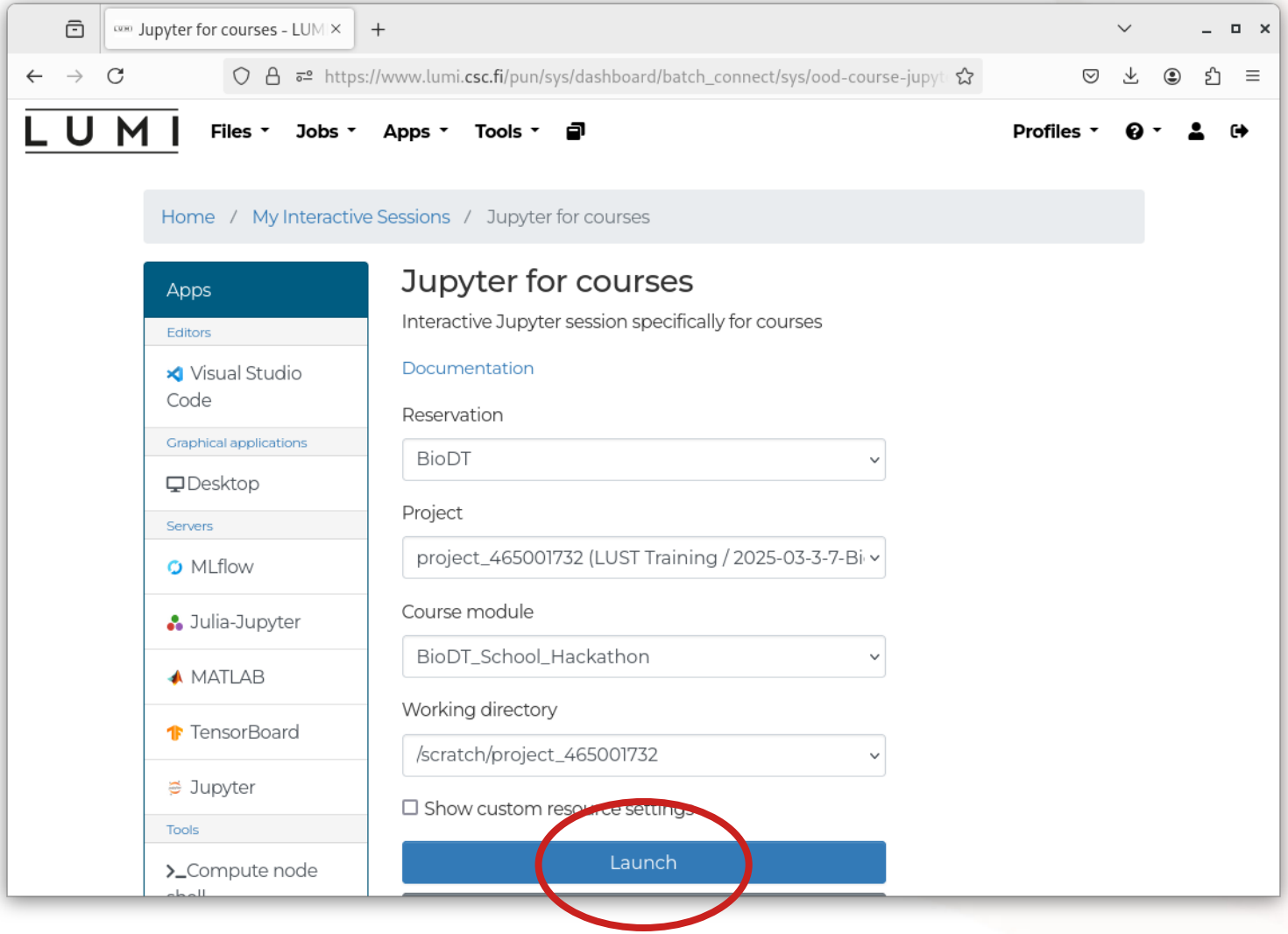
Launching course Jupyter notebook in LUMI web interface

- Open https://www.lumi.csc.fi, click “Go to login” → Select MyAccessID
- Click “Jupyter for courses”



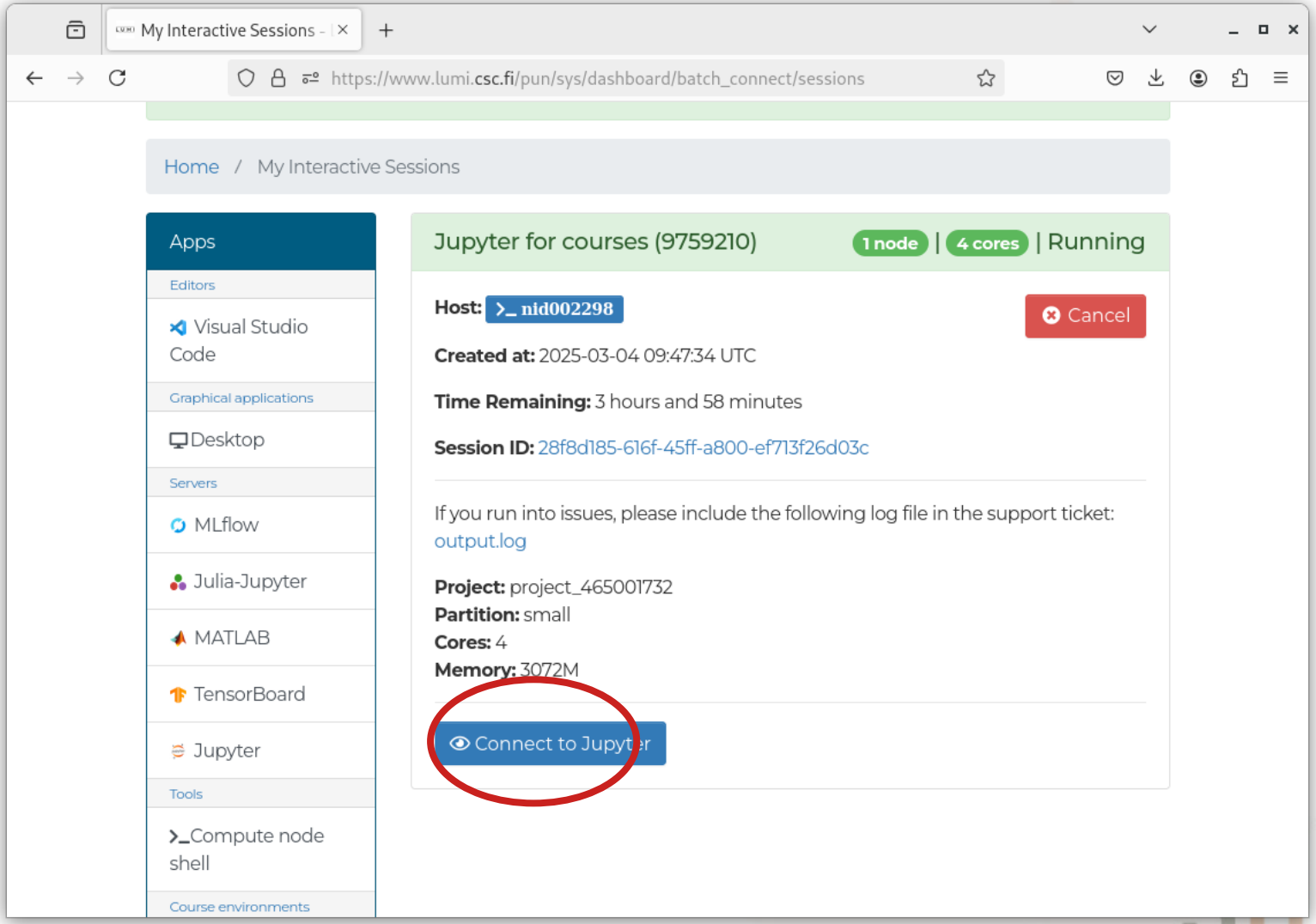
Launching course Jupyter notebook in LUMI web interface

- Click “Launch”



Launching course Jupyter notebook in LUMI web interface

- Wait and then click “Connect to Jupyter”



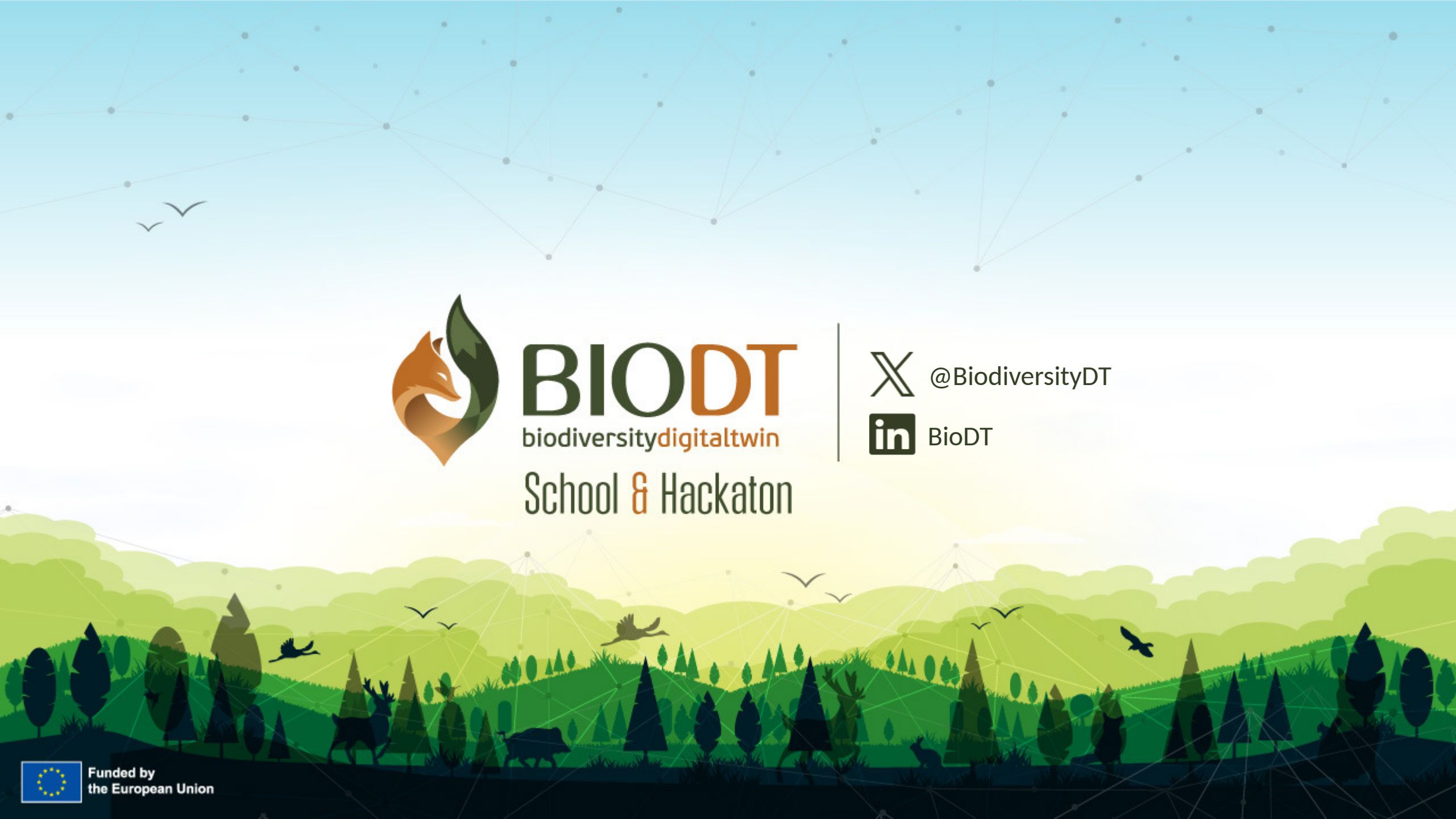
BIODT
biodiversitydigitaltwin
School & Hackaton

Using LUMI supercomputer beyond the School & Hackathon

LUMI Supercomputer - Get Started

- LUMI is a EuroHPC supercomputer
  - Available for all users in Europe
- Users in LUMI consortium countries can access the national LUMI resources in addition

- To get started, apply a project
  - https://lumi-supercomputer.eu/get-started/


BIODT
biodiversitydigitaltwin
School & Hackaton
@BiodiversityDT
BioDT
Funded by
the European Union