Scalable Data Analysis with Dask

Lyda Hill Department of Bioinformatics
Kevin M. Dean, Conor McFadden

August 6, 2025

Nanocourse Outline

- **09:00 – 10:00** – Session 1 - Introduction to Dask and Parallel Computing
- **10:15 – 11:00** – Session 2 - Dask Arrays for Big Image Data
- **11:00 – 12:00** – Lab 1 - Processing Large Image Data with Dask
- **12:00 – 1:00** – Lunch Break
- **1:00 – 1:30** – Session 3 - Dask DataFrames for Large Tabular Data
- **1:30 – 2:15** – Lab 2 - Analyzing Large Tabular Data with Dask
- **2:30 – 3:15** – Session 4 - Distributed Dask: Scaling to Clusters with SLURM
- **3:30 – 4:15** – Lab 3 - Adapting Code for Distributed Dask
- **4:15 – 4:30** – Wrap-Up, Q&A, and Next Steps

Introduction to Dask and Parallel Computing

Session 1: 9:00 - 10:00 AM

Big Data Challenges

- **Data Volume:** Modern datasets (microscopy images, genomics); often exceed a single machine's RAM, making traditional in-memory tools (e.g., Pandas); impractical. Even data close to memory size can be unwieldy due to copies during processing.
- **CPU Limitations:** Python's Global Interpreter Lock (GIL) means pure Python code runs on only one core, so typical NumPy/Pandas workflows run single-threaded and fail to use modern multi-core CPUs. This leaves significant performance potential untapped on multi-core machines.
- **Need for Parallelism:** To handle big data within reasonable time, we must go *out-of-core* (process data in chunks from disk) and exploit parallel computing—using all CPU cores or multiple cluster nodes in tandem. This requires tools beyond the standard single-threaded Python approach (e.g., multi-processing or distributed computing frameworks).

Python’s Global Interpreter Lock

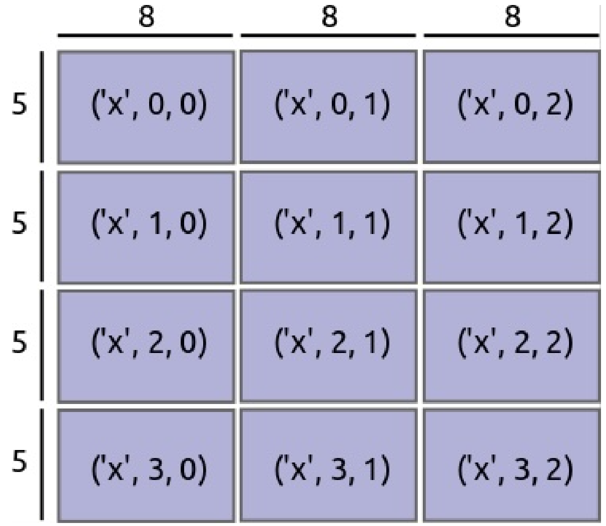
Python’s Global Interpreter Lock (GIL) is a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecodes simultaneously. This leaves significant performance potential untapped on multi-core machines. For example…

```python
from time import sleep

data = [1, 2, 3, 4, 5, 6, 7, 8]

def inc(x):
    sleep(1)
    return x + 1


results = []
for x in data:
    y = inc(x)
    results.append(y)

total = sum(results)
```

Dask Overview - Out-of-Core & Lazy Execution

Blocked Algorithms: Dask breaks up large data into many small pieces (chunks) that fit in memory. This enables computations on datasets larger than RAM by processing each chunk sequentially or in parallel (out-of-core execution).

Lazy Evaluation: Operations on Dask collections are *lazy* – they build a task graph of work to be done, but no actual computation happens until you explicitly call `.compute()` or similar to execute the graph. This avoids needless work and allows optimization before running tasks.



Dask Overview - Out-of-Core & Lazy Execution

Task Graph & Scheduler: Each Dask computation creates a directed acyclic graph (DAG) of tasks. When triggered, Dask's *scheduler* orchestrates executing these tasks on parallel hardware (threads, processes, or cluster workers) to produce the result. This separation of the task graph from execution allows Dask to scale transparently.

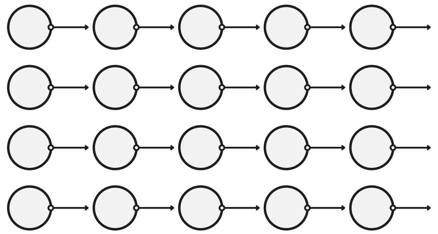
What is a DAG?: A DAG is a graph structure with directed edges and no cycles. In Dask, each node represents a task, and edges represent dependencies between tasks.

- A task is a unit of work (e.g., a function call), and dependencies indicate which tasks must complete before others can start.
- Tasks are automatically scheduled based upon their dependencies, allowing Dask to execute them in parallel where possible.

High-Level Abstraction: User Collections (e.g. Dask Arrays, DataFrames, etc) lets users write high-level code while Dask handles parallel execution.

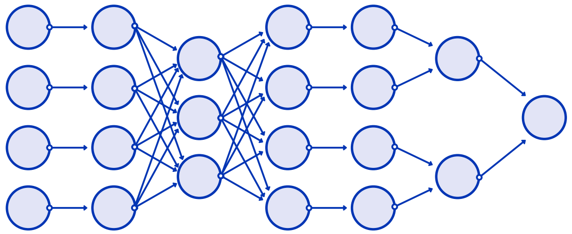
Embarrassingly Parallel

Hadoop/Spark/Dask/Airflow/Prefect



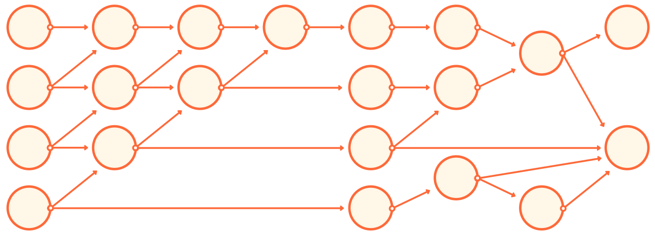
MapReduce

Hadoop/Spark/Dask



Full Task Scheduling

Dask/Airflow/Prefect



Parallelism on a Single Machine

- **Multi-Threading:** Dask's default scheduler on a single machine uses a thread pool to execute tasks concurrently. This incurs very little overhead (tasks run in the same process), and works well when computations release the GIL (e.g. NumPy, Pandas C code). Due to Python's GIL, multi-threading yields speedups mainly for numeric or I/O-bound workloads.

- **Multi-Processing:** For Python-heavy workloads, Dask can use multiple processes. The multiprocessing scheduler runs tasks in separate Python processes, bypassing the GIL to achieve true parallelism. This allows parallel execution even for code that doesn't release the GIL (e.g. processing text or Python objects). The trade-off is overhead: transferring data between processes can be costly, especially if large data needs to be serialized.

Choosing a Scheduler: Dask chooses a sensible default (threads for arrays/dataframes, processes for bags) but you can configure the scheduler. For compute-heavy pure Python tasks, using processes (or the distributed scheduler in local mode) can improve performance. For numeric array/dataframe tasks, threads are often effective since NumPy/Pandas do the heavy lifting in C/C++ (no GIL contention).

Dask’s Scheduler System

- **Advanced Local Scheduler:** Dask’s distributed scheduler can run on a single machine or across a cluster. Even on one machine, many users prefer the distributed scheduler for its richer features: an asynchronous Future API, a live diagnostic dashboard, and smarter task scheduling that can outperform the simple multiprocessing approach.
- **Scaling to Clusters:** The distributed scheduler shines when scaling out to multiple machines. It can coordinate a cluster of worker processes spread across several nodes, handling communication and load balancing. Dask can integrate with HPC job schedulers (SLURM, PBS, LSF, etc.) to launch workers on a cluster. This means you can write your code using Dask’s API and then deploy it on an HPC cluster, harnessing hundreds of cores or more.

Dask’s Scheduler System

Same Code, Larger Scale…

- Thanks to Dask’s abstraction, code written for Dask on your laptop can run on a cluster with minimal or no changes.
- The scheduler and cluster take care of distributing data and computations.
- This lets you prototype locally and then scale out to big data on HPC when needed – a key benefit for scientific computing workflows.

```python
from dask.distributed import Client, LocalCluster

# Local cluster with custom configuration
cluster = LocalCluster(n_workers=4, threads_per_worker=2, memory_limit='2GB')
client = Client(cluster)
```

Dask Collections (Parallel Data Structures)

Dask provides high-level collections that mimic familiar APIs, allowing you to scale up workflows with minimal code changes:

- **Dask Array:** Parallel N-dimensional array (NumPy-like). Useful for large numerical data (e.g. large images, multi-dim arrays).
- **Dask DataFrame:** Parallel DataFrame (Pandas-like). For large tabular datasets (e.g. CSVs, data frames) split into many partitions.
- **Dask Bag:** Parallel collection for unstructured data (like a list of Python objects, text logs, JSON records). It's a general-purpose container.
- **Dask Delayed:** A low-level way to build custom task graphs. You wrap arbitrary Python functions with dask.delayed to create tasks and dependencies manually.
- **Dask Futures:** An asynchronous programming interface for real-time task execution and result retrieval, similar to Python's `concurrent.futures`.

All these collections run on the same Dask scheduler infrastructure. They mirror NumPy/Pandas APIs closely, so you can often switch to Dask by changing imports and adding a `.compute()` call.

Minimal Code Changes: Pandas vs Dask DataFrame

For example, switching from Pandas to Dask DataFrame in an analysis is straightforward – usually just change the import, possibly load data in partitions, and call .compute() when needed:

```python
# Pandas usage (single-machine, in-memory)
import pandas as pd
df = pd.read_csv("data.csv")
result = df.groupby('category')['value'].mean()

# Dask usage (parallel, can be larger-than-memory)
import dask.dataframe as dd
df = dd.read_csv("data_*.csv")        # can read multiple files in chunks
result = df.groupby('category')['value'].mean().compute()
```

Both snippets above do the same group-by operation. The Dask version spreads the work across many partitions and CPUs, then uses .compute() to get the final result.

Minimal Code Changes: NumPy vs Dask Array

For NumPy arrays, switching to Dask Array is similarly straightforward:

```python
# NumPy usage (single-machine, in-memory)
import numpy as np
arr = np.random.rand(10000, 10000)  # large array
result = np.mean(arr, axis=0)

# Dask usage (parallel, can be larger-than-memory)
import dask.array as da
arr = da.random.random((10000, 10000), chunks=(1000, 1000))  # large array in chunks
result = da.mean(arr, axis=0).compute()  # compute mean across chunks
```

Both snippets perform the same mean operation on a large array. The Dask version processes the data in chunks, allowing it to handle arrays larger than memory and utilize all CPU cores.

exercises/session_1/exercise_0.ipynb

Automated Scheduling vs Manual Scheduling

Dask Delayed for Custom Task Graphs

Standard Automated Scheduling (Dask Arrays/DataFrames)

- Automatic chunking: Dask automatically breaks data into chunks
- Predefined operations: Task graphs are built from known operations (groupby, mean, etc.)
- Implicit parallelization: You don't specify individual tasks - Dask figures out the parallel structure

Dask Delayed - Manual Task Graphs

- Explicit task creation: You manually wrap functions to create tasks
- Custom dependencies: You define which tasks depend on which others
- Arbitrary workflows: Can handle any Python function, not just array/dataframe operations

Demo: Parallelizing with Dask Delayed

Here two independent tasks can run in parallel, then feed into an add task which combines their results.

```python
import numpy as np
from dask import delayed
from dask.distributed import Client

client = Client()  # Launch local cluster of workers

# Two large arrays (e.g. parts of a dataset)
A, B = np.random.random((10000, 10000)), np.random.random((10000, 10000))

# Create delayed tasks for summing each array
sumA, sumB = delayed(np.sum)(A), delayed(np.sum)(B)

# Create another task to add the two sums
total = delayed(lambda x, y: x + y)(sumA, sumB)
result = total.compute()
print(result)
```

The two `np.sum` operations run concurrently on separate workers, then the final addition combines the results.

exercises/session_1/exercise_1.ipynb

Dask Arrays for Big Image Data

Session 2: 10:15 - 11:00 AM

The Problem with Tiff Files

Logical Structure: TIFF files can store single or multi-page images. Multi-page TIFF include multiple images within a single file, typically sequentially.

On-Disk Representation:

- Image planes typically stored as contiguous blocks or sequential slices in one large file.
- 3D regions of interest must be read from non-contiguous slices, which is inefficient.
- Metadata (e.g., image dimensions, data type) stored in headers as XML.

Parallel Read/Write:

- Reading TIFF files in parallel is challenging due to sequential data layout; often requires reading headers to locate slices.
- Limited parallel I/O, leading to bottlenecks for large images.

Dask Array Storage

Logical Structure: Dask arrays themselves are an abstraction without a built-in storage format. Instead, they use external chunked storage backends like Zarr or HDF5 for persistent storage.

On-Disk Representation (Zarr):

- Each chunk is stored as an individual compressed binary file within a hierarchical directory structure (e.g., /array_name/0.0.0, /array_name/0.0.1, etc.).
- A JSON metadata file describes array shape, chunk size, data type, compression.

Parallel Read/Write:

- Each chunk can be accessed independently, facilitating highly parallel I/O.
- Excellent performance in distributed environments or cloud storage.
- Compression of each chunk performed independently, reducing storage footprint.

Alternatives to Dask Arrays

Xarray: Built on top of NumPy and Dask, Xarray adds labeled dimensions and coordinates to arrays, making it ideal for multidimensional scientific data. Especially popular in atmospheric, oceanographic, and environmental sciences.

N5/Zarr: A storage format designed for chunked, compressed, and parallel data access. Often used alongside Dask and Xarray for storing and handling large datasets efficiently. Zarr enables efficient reading and writing of large-scale datasets, particularly beneficial in cloud and HPC environments.

HDF5: A widely used file format for storing large amounts of numerical data. HDF5 supports chunking and compression, making it suitable for large datasets. It is often used in scientific computing and data analysis.

OME-NGFF: A format specifically designed for biological imaging data, providing a standardized way to store and share image metadata. OME-NGFF is built on top of Zarr and HDF5, and is optimized for cloud computing environments.

Xarray, Zarr, and OME-NGFF have metadata specifications that accommodate data types associated with image analysis. Outputs, such as segmentation labels, can be saved in a common data structure with the original pixel data and metadata, providing provenance.

Object Storage vs. Traditional File Systems

Cloud Object Storage:

- Small objects distributed across multiple servers
- Entire binary objects read as monolithic blocks
- Higher latency but greater parallelization potential
- No random access within files - must read complete objects

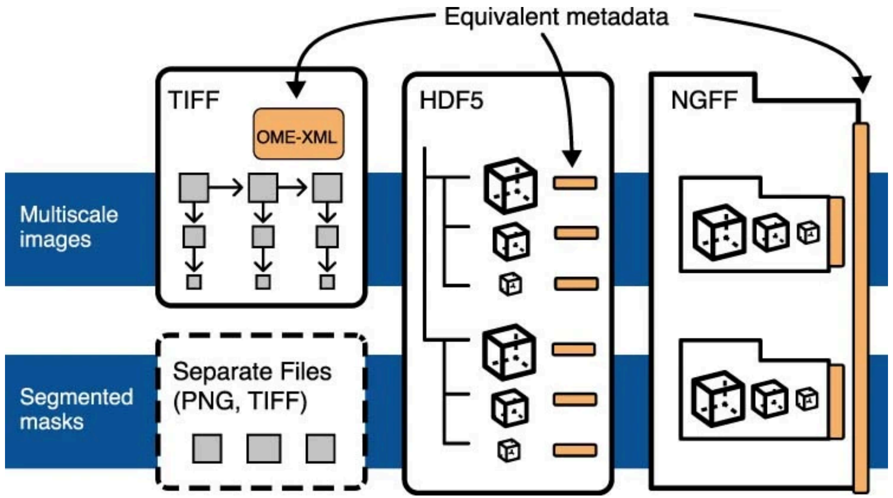
Local HPC Environments:

- Traditional file systems with random access capabilities
- Lower latency, direct memory mapping possible
- Limited by single-node I/O bandwidth
- Can seek to specific byte ranges within large files

Converting Image Formats

Universal Metadata Standard

- Open Microscopy Environment XML schema provides standardized metadata for biological imaging
- Describes image dimensions, acquisition parameters, experimental conditions
- Same OME-XML can be used across different storage backends



Converting Image Formats

Bio-Formats (Java/Python): Reads 150+ proprietary formats, writes OME-TIFF, OME-NGFF, etc.

- `bioformats2raw` : Proprietary → OME-NGFF/Zarr
- `raw2ometiff` : OME-NGFF → OME-TIFF
- `python-bioformats` : Python wrapper for Bio-Formats

Python Ecosystem:

- `tifffile` : TIFF ↔ NumPy arrays with metadata
- `zarr-python` : Create/read Zarr arrays
- `ome-zarr-py` : OME-NGFF specific operations
- `h5py` : HDF5 file I/O

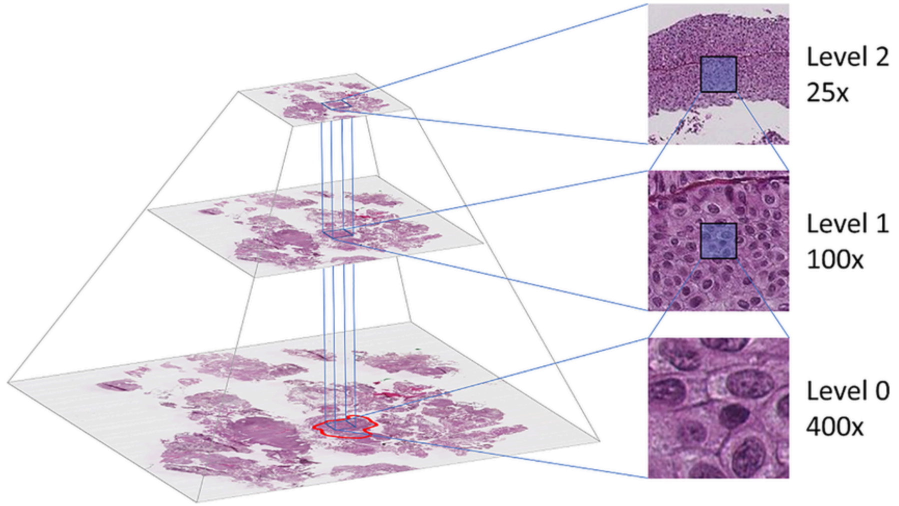
```
# Proprietary → OME-TIFF
bfconvert input.lsm output.ome.tiff

# OME-TIFF → OME-NGFF
bioformats2raw input.lsm output.zarr
```

Multi-Resolution Image Pyramids

- Store same image at multiple resolution levels (scales)
- Full resolution at level 0, each subsequent level downsampled by factor of 2
- Load only the resolution needed for current viewing/analysis task

Advantages:

- Quick overview at low resolution, zoom to full detail on demand
- Avoid loading massive full-resolution data for overview tasks



Optimizing Chunk Sizes

Critical for performance in Dask Arrays and Zarr storage

Chunk Size Trade-offs.

- **Too Small:** Task scheduling overhead (~1ms per task) dominates runtime
- **Too Large:** Memory overflow and reduced parallelism
- **Sweet Spot:** 10MB - 1GB chunks, tasks taking >100ms each

Multi-Dimensional Chunking Strategies

- **Spatial Chunking (X,Y,Z):** Best for pixel-wise operations
- **Temporal Chunking (T):** Efficient for time-series analysis
- **Hybrid Chunking:** Balance multiple dimensions

```
# Generate multi-resolution pyramid with custom chunk size
bioformats2raw --resolutions 5 --tile_width 512 --tile_height 512 --chunk_depth 1 input.czi output.zarr

# Specify custom downsampling factors with optimized chunks
bioformats2raw --resolutions 4 --downsample-type AREA --tile_width 1024 --tile_height 1024 input.lsm pyramid.zarr
```

Optimizing Chunk Sizes

Critical for performance in Dask Arrays and Zarr storage

Matching Chunks to Analytical Operations

- **Feature Detection:** Chunks should be 3-5× larger than the largest feature you're detecting
- **Convolution/Filtering:** Include padding equal to filter kernel size. Prevents edge effects
- **2D Operations on 3D Data:** Use thin Z-chunks (1-3 planes) with larger XY dimensions
- **Time Series Analysis:** Chunk along time dimension for operations across timepoints

Lazy Loading of Large Image Data

Loading Zarr as Dask Array

```python
import dask.array as da

# Load Zarr dataset as Dask array
arr = da.from_zarr('data.zarr')

# Or specify specific array within Zarr group, such as the resolution level.
# e.g., arr = da.from_zarr('data.zarr', component='0')

# Inspecting Array Structure
print(f"Shape: {arr.shape}")           # (10, 65, 1024, 1024)
print(f"Chunks: {arr.chunks}")         # ((5, 5), (65,), (512, 512), (512, 512))
print(f"Dtype: {arr.dtype}")           # uint16
print(f"Size: {arr.nbytes / 1e9:.2f} GB")  # 10.22 GB

# Rechunk for different access patterns
spatial_chunks = arr.rechunk((1, 1, 512, 512))      # Spatial operations
temporal_chunks = arr.rechunk((10, -1, 256, 256))   # Time-series analysis
balanced_chunks = arr.rechunk((2, 32, 256, 256))    # Balanced approach

# Persist optimized chunks to storage
balanced_chunks.to_zarr('optimized.zarr')
```

exercises/session_2/exercise_2.ipynb

Dask Array Operations

map_blocks: Element-wise Operations

Apply Functions to Array Chunks Independently

- Process each chunk separately without communication between chunks
- Ideal for pixel-wise operations (filtering, thresholding, normalization)
- Maintains chunk structure and enables perfect parallelization

Key Points

- No overlap between chunks - perfect for independent operations
- Maintains original chunking structure
- Minimal memory

Dask Array Operations

map_blocks: Element-wise Operations

```python
import dask.array as da
import numpy as np
from skimage import filters

arr = da.from_zarr('microscopy_data.zarr', component='0/0')

def gaussian_filter_chunk(chunk, sigma=2):
    return filters.gaussian(chunk, sigma=sigma, preserve_range=True)

# Process all chunks in parallel
filtered = da.map_blocks(
    gaussian_filter_chunk, # Function to apply
    arr, # Input Dask array
    sigma=2,
    dtype=arr.dtype,
    meta=np.array((), dtype=arr.dtype))

result = filtered.compute()
```

Dask Array Operations

Dask map_overlap: Neighborhood Operations

Handle Operations Requiring Neighboring Pixels

- Automatically manages overlap between chunks for convolution-like operations
- Essential for edge detection, morphological operations, and local feature analysis
- Handles boundary conditions and chunk edge artifacts

Key Parameters:

- depth: Overlap size (pixels/voxels) - must accommodate operation’s neighborhood
- boundary: How to handle array edges ('reflect', 'constant', 'wrap')
- Automatically trims overlap from final result

Dask Array Operations

Dask map_overlap: Neighborhood Operations

```python
import dask.array as da
import numpy as np
from skimage import morphology, feature

# Load imaging data
arr = da.from_zarr('cell_segmentation.zarr', component='0/0')

# Edge detection requires neighboring pixels
def sobel_edge_detection(chunk):
    return feature.canny(chunk, sigma=1.0)

# Apply with overlap to handle chunk boundaries
edges = da.map_overlap(
    sobel_edge_detection,
    arr,
    depth=10,          # 10-pixel overlap on all sides
    boundary='reflect', # Handle array edges
    dtype=bool,
    meta=np.array((), dtype=bool)
)

# Compute results
edge_map = edges.compute()
```

Dask Array Operations

blockwise: Advanced Multi-Array Operations

```python
import dask.array as da
import numpy as np

fluorescence = da.from_zarr('fluorescence.zarr', component='0/0')
brightfield = da.from_zarr('brightfield.zarr', component='0/0')
background = da.from_zarr('background.zarr', component='0/0')

def ratio_calculation(fluor_chunk, bright_chunk, bg_chunk):
    corrected_fluor = fluor_chunk - bg_chunk
    corrected_bright = bright_chunk - bg_chunk
    ratio = np.divide(
        corrected_fluor, corrected_bright, out=np.zeros_like(corrected_fluor), where=corrected_bright!=0)
    return ratio

ratio_image = da.blockwise(
    ratio_calculation,           # Function to apply
    'ijk',                      # Output dimensions
    fluorescence, 'ijk',        # First input array and its dimensions
    brightfield, 'ijk',         # Second input array and its dimensions
    background, 'ijk',          # Third input array and its dimensions
    dtype=np.float32,
    concatenate=False
```

Dask Array Operations

blockwise: Advanced Multi-Array Operations

```python
import dask.array as da
import numpy as np

# Mathematical operations with different array shapes
mask = da.from_zarr('cell_mask.zarr')  # Shape: (1024, 1024)
timeseries = da.from_zarr('timelapse.zarr')  # Shape: (100, 1024, 1024)

def apply_mask(time_chunk, mask_chunk):
    """Apply 2D mask to each timepoint"""
    return time_chunk * mask_chunk[np.newaxis, :, :]

masked_series = da.blockwise(
    apply_mask, 'tij',
    timeseries, 'tij',
    mask, 'ij',
    dtype=timeseries.dtype
)
masked_result = masked_series.compute()
```

Dask Array Operations

blockwise: Advanced Multi-Array Operations

Element-wise Operations Across Multiple Arrays

- Apply functions that operate on corresponding chunks from multiple arrays
- More flexible than `map_blocks` for operations involving multiple inputs
- Handles broadcasting and alignment automatically
- Perfect for mathematical operations between different datasets

Key Features:

- Operates on multiple Dask arrays simultaneously
- Preserves chunking structure across all inputs
- Supports custom output chunk shapes and data types
- Handles different array shapes through broadcasting

Dask Array Operations

Chaining Operations

Build Complex Processing Pipelines

- Chain multiple `map_blocks` operations to create computational graphs
- Only one `compute()` call triggers the entire pipeline
- Dask optimizes the task graph for efficient execution

Advantages:

- Minimizes memory usage - intermediate arrays stay as task graphs
- Optimized execution plan across the entire pipeline
- Easy to modify pipeline without recomputing previous steps

Dask Array Operations

Chaining Operations

```python
import dask.array as da
import numpy as np
from skimage import filters, exposure, morphology

arr = da.from_zarr('microscopy_data.zarr', component='0/0')

def gaussian_filter_chunk(chunk, sigma=2):
    return filters.gaussian(chunk, sigma=sigma, preserve_range=True)

def normalize_chunk(chunk):
    return exposure.rescale_intensity(chunk, out_range=(0, 1))

def threshold_chunk(chunk, threshold=0.5):
    return chunk > threshold

def morphology_chunk(chunk):
    return morphology.remove_small_objects(chunk, min_size=50)

# Chain operations
filtered = da.map_blocks(gaussian_filter_chunk, arr, sigma=1.5, dtype=arr.dtype)
normalized = da.map_blocks(normalize_chunk, filtered, dtype=np.float32)
binary = da.map_blocks(threshold_chunk, normalized, threshold=0.3, dtype=bool)
```

Other Dask Array Operations

Useful Operations

da.stack and da.concatenate

- Combining multiple arrays along new or existing dimensions

da.reduction operations

- sum(), mean(), max(), std() across dimensions. Tree reduction patterns for efficient parallel computation

da.apply_gufunc

- Generalized universal functions for more complex array operations, handles operations that require custom broadcasting or shape manipulation

da.overlap module

- overlap_internal for custom neighborhood operations, more control than map_overlap for advanced use cases

Processing Large Image Data with Dask

Lab 1: 11:00 – 12:00

Lab 1

Part 1 - Processing Large Image Data with Dask

- Load an image series into dask as a Dask array. Evaluate its shape, dtype, and chunk sizes.
- Perform reductions like max/mean projections across the image stack, and plot the results.
- Apply a Gaussian filter to each image slice using map_blocks or map_overlap. Compare the resulting image data.
- Rechunk the Dask array and evaluate the performance of the operation.
- Save results to Zarr: Store the large filtered Dask array (or the computed projection if space is limited) in Zarr format.
- Compare the results with a standard numpy/scikit-image workflow to see the performance difference.

Note: Can be performed on your local machine with a mapped network drive or on BioHPC.

Data:

- `/archive/shared/MIL/2d.zarr`
- `/archive/shared/MIL/3d.zarr`

Lab 1

Part 2 - Creating a Custom Dask Routine

The Otsu thresholding algorithm is a common technique for image segmentation. In this lab, you will implement a custom Dask routine to apply Otsu’s method across a large image dataset. The identified threshold should be the global threshold, and not applied on a per-chunk basis.

Note: Can be performed on your local machine with a mapped network drive or on BioHPC.

Data:

- `/archive/shared/MIL/2d.zarr`
- `/archive/shared/MIL/3d.zarr`

Lab 1: Key Takeaways

- **Parallel, Larger Than Memory Analysis:** Dask Array enables parallel, out-of-core computation on large images. It breaks big arrays into chunks and processes them with blocked algorithms, so you can work with datasets larger than memory.
- **Chunk wisely:** The size and shape of chunks matters. Too many tiny chunks add overhead (each task has scheduling cost ~1ms), while huge chunks won't fit in RAM. Aim for a sweet spot in between – chunks that are as large as possible but still memory-friendly (often tens of MBs each).
- **Lazy loading and compute:** Dask integrates easily with file-based data. You can load image slices lazily and only compute what you need. Operations (slicing, projections, filtering) run per chunk and aggregate, so you never have to load the entire dataset into memory.
- **Zarr format:** a great option for storing big arrays. It keeps data chunked on disk, which pairs perfectly with Dask's chunked processing. With Zarr, you can save intermediate results or datasets and reload them efficiently later – Dask will only read the chunks required for your analysis.

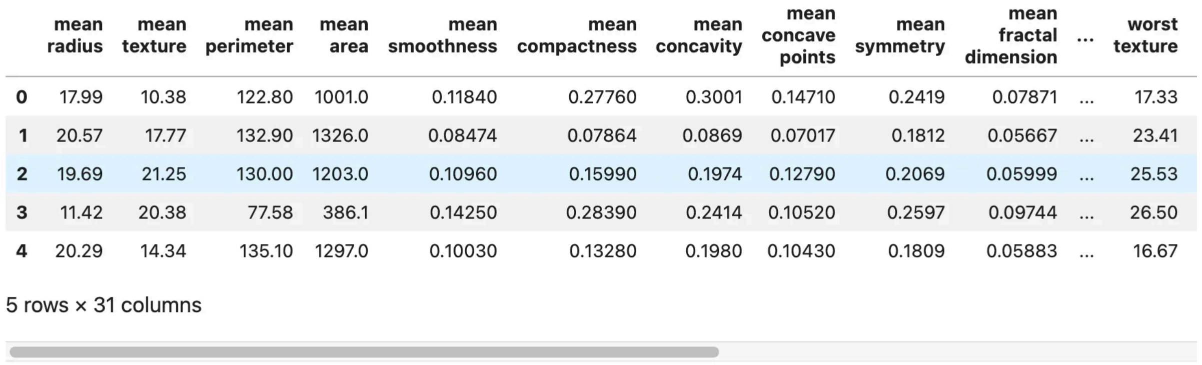
Dask DataFrames for Large Tabular Data

Session 3: 1:00 – 1:30 PM

Refresher: What is a Pandas DataFrame?

- A **Pandas DataFrame** is a 2D, in-memory table of labeled, heterogeneous data.
- It's built on top of NumPy and designed for **fast, interactive data analysis**.
- Ideal for small to medium datasets that **fit entirely in RAM**.
- Provides rich functionality: filtering, joining, grouping, reshaping, time series support, etc.

```python
import pandas as pd

df = pd.read_csv("data.csv") # e.g. sklearn breast_cancer dataset ...

df.head()
```



	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry	mean fractal dimension	...	worst texture
0	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	0.2419	0.07871	...	17.33
1	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	0.1812	0.05667	...	23.41
2	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	0.2069	0.05999	...	25.53
3	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	0.2597	0.09744	...	26.50
4	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	0.1809	0.05883	...	16.67

5 rows × 31 columns

What is a Dask DataFrame?

- A **Dask DataFrame** is a parallel, out-of-core version of a Pandas DataFrame.
- It supports many familiar Pandas operations: `groupby`, `filter`, `join`, etc.
- Operations are **lazy**, building a task graph for optimized execution.

```python
import dask.dataframe as dd

df = dd.read_csv("data/large_dataset_*.csv")
```

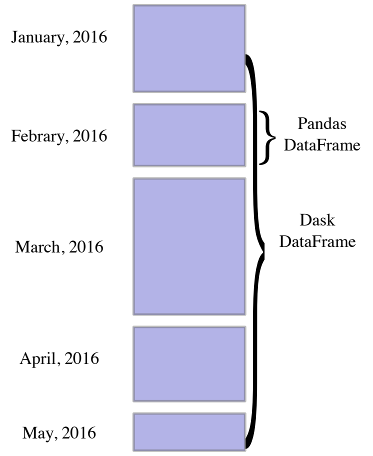
Why Not Use Pandas?

- **Single-threaded**: Pandas operates on one core – no automatic parallelism.
- **In-memory only**: Entire dataset must fit in RAM.
- **Eager execution**: Each line runs immediately, limiting optimization opportunities.
- **Limited scalability**: Great for small data, but struggles with "big-enough" data.

How Dask DataFrames Work

- A **Dask DataFrame** is made of many smaller Pandas DataFrames called **partitions**.
- Operations are applied **independently** to each partition.
- Instead of executing immediately, Dask builds a **task graph** and delays execution until `.compute()` is called.
- This enables **parallel**, **out-of-core**, and **optimized execution**.

```python
# Nothing runs yet — this is lazy
result = df[df.score > 0.9].groupby("region")["value"].mean()

# Now the computation happens
result.compute()
```



Reading Data with Dask DataFrame

- In Pandas you would `read_csv` one file:

```python
import pandas as pd

df = pd.read_csv("./longitudinal_data/2025-07-08.csv")
```

- In Dask, you can similarly `dd.read_csv()` to lazily load one or many files with a wildcard `*`.

```python
import dask.dataframe as dd

# Read multiple CSVs with wildcard
df = dd.read_csv("./longitudinal_data/2025-*.csv")
```

- Files are **not read into memory immediately** – just metadata is loaded.
- You can also read **Parquet**, **JSON**, and data from **S3, GCS**, etc.
- Running this may yield a `RuntimeError`:

```
ImportError: Missing optional dependency 'pyarrow'.  Use pip or conda to install pyarrow.
```

- Dask reads data in **blocks** across partitions using **file-access backends**, like `pyarrow`. Pip install it!

Inspecting Partitions in Dask DataFrames

- When you load multiple files with Dask splits the dataset into **partitions**.
- Each partition is just a **Pandas DataFrame**, held separately in memory or read lazily from disk.

```python
df.npartitions  # We used "2025-*.csv" as our wildcard, so n = 12 chunks
```

```
12
```

```python
# you can access the partitions individually
print(df.partitions[0])
```

```
Dask DataFrame Structure:
                        Date      ID     Name TumorDiameter_cm     Drug
npartitions=1
                      string   int64   string          float64   string
                         ...     ...      ...              ...      ...
Dask Name: partitions, 3 expressions
Expr=Partitions(frame=ArrowStringConversion(
    frame=FromMapProjectable(7d94751)),
    partitions=[0])
```

- Note that partitions are just lazy expressions referencing each file: **must call** `df.partitions[0].compute()` **!**

Lazy Execution: Nothing Happens Until `.compute()`

- Dask DataFrames don’t load or compute data immediately.
- Operations build a **task graph** – a deferred plan, not actual results.
- You can slice a partition, filter rows, or chain operations – all of it stays lazy until `.compute()`.

```python
df = dd.read_csv("./longitudinal_data/2025-*.csv")          # No files read yet

df_filtered = df[df.Drug == "Placebo"]
print(type(df_filtered)) # Still lazy
# OUTPUT: <class 'dask.dataframe.dask_expr._collection.DataFrame'>

df_computed = df_filtered.compute()    # Now Dask loads and computes
print(type(df_computed))
# OUTPUT: <class 'pandas.core.frame.DataFrame'> We're back to Pandas!

print(df_computed.head())
```

```
          Date    ID       Name  TumorDiameter_cm     Drug
3   2025-01-08  1618     Elijah             0.396  Placebo
4   2025-01-08   663      James             0.180  Placebo
12  2025-01-08  1927      Ethan             0.519  Placebo
18  2025-01-08   161  Sebastian             0.319  Placebo
19  2025-01-08   291      Mateo             0.247  Placebo
```

Filtering, Grouping, and Aggregating

- Dask supports many familiar Pandas operations – but they remain **lazy** until `.compute()` is called.
- Example: Filter the dataset, group by drug, and calculate the mean tumor diameter.

```
# Load and filter patients
df = dd.read_csv("./longitudinal_data/2025-*.csv")
filtered = df[df.TumorDiameter_cm > 0.3]

# Group and aggregate
summary = filtered.groupby("Drug")["TumorDiameter_cm"].mean()

# Trigger the computation
summary.compute()
```

```
Drug
Lunadrex    0.412
Placebo     0.489
Velocor     0.317
Zanthera    0.371
Name: TumorDiameter_cm, dtype: float64
```

Applying Functions to Partitions (mapping)

- Use `df.map_partitions()` to apply a custom function to each partition (a Pandas DataFrame).
- Great for:
  - Row-wise operations not built into Dask
  - Applying Pandas logic in parallel
  - Custom formatting or filtering

```python
def double_tumor(df):
    df["Tumor_x2"] = df["TumorDiameter_cm"] * 2
    return df

new_df = df.map_partitions(double_tumor)

new_df.head()
```

```
         Date    ID    Name  TumorDiameter_cm      Drug  Tumor_x2
0  2025-01-08  1409    Liam             0.290  Lunadrex     0.580
1  2025-01-08   328    Noah             0.579  Lunadrex     1.158
2  2025-01-08   151  Oliver             0.470  Lunadrex     0.940
3  2025-01-08  1618  Elijah             0.396   Placebo     0.792
4  2025-01-08   663   James             0.180   Placebo     0.360
```

Example: Plotting Tumor Diameter over all Patients

- Let’s use Dask DataFrame to plot average tumor progression over time spread over many files.
- We want to **group**, **mean**, **normalize** and **plot**:
  - **What can Dask do? What can it *NOT* do?**

Why not cram everything before `.compute()`?

Dask DataFrames can’t parallelize tasks that require **Global Context**!

Why `.pivot()` doesn’t work:

- Pivoting needs to **know all unique values** in the pivot column (e.g. `"Drug"`).
- But those values could be spread across different files/partitions.
- Dask **can’t collect those values** without first reading all the data.

Why normalization by first month doesn’t work:

- `.iloc[0]` only makes sense when the full DataFrame is in memory.
- Dask has no concept of “first row overall” across all partitions.

So what can we do?

- Let Dask do **Grouping**, **Averaging**, and **Index Resets** in parallel.
- Then use Pandas for global, in-memory logic like **Normalizing**, **Pivoting**

Summary: Dask DataFrames

- Dask lets you work with **big tabular data** using familiar Pandas-like code.
- It operates **lazily**, building a task graph – nothing runs until you `.compute()`.
- Data is split into **partitions**, enabling parallelism and out-of-core processing.
- **Partitions** under the hood are just Pandas `DataFrames`.
- **Careful!** Some global operations (like `.pivot()`, plotting, `.iloc[]`) cannot be parallelized. **Use Pandas!**

Dask DataFrames for Large Tabular Data

Lab 2: 1:30 – 2:15 pm

Lab 2: Analyzing Large Tabular Data with Dask

- Use the first code block to generate some patient tumor growth data and save it locally as CSV. You can play with some of the parameters.
- Start up a Dask Client to visualize the Task Stream while working with Dask DataFrames.
- Explore simple Pandas operations with Dask DataFrames, including `map_partitions`.
- Try to combine what we've learned to recreate the average tumor growth, but now try to include **error bars**. Pandas Aggregation (`.agg()`) can be useful for this. You can use the dashboard to visualize the task stream.

Note: Can be performed on your local machine or on BioHPC.

Distributed Dask: Scaling to Clusters with SLURM

Session 4: 2:30 – 3:15 PM

Clusters versus Runners

Clusters: Creates new jobs for Dask workers. Scales to the number of workers needed, if available.

- Can request a fixed number of resources, or dynamically scale number of workers with project demands.
- Requires job queue permissions
- Better for interactive/adaptive workloads

Runners: Uses existing allocated resources to run Dask workers. Does not scale beyond the initial allocation.

- Uses existing resources without submitting new jobs.
- Sets up scheduler on head node, workers on remaining nodes

Key Features:

- Dask automatically discovers available nodes from SLURM environment
- Dask handles network configuration and port management

Dask Cluster Types

HPC Clusters

Multi-Machine Parallelism

- SLURM, PBS, LSF, SGE support
- Submit Dask workers as batch jobs
- Automatic scaling based on workload

```python
from dask_jobqueue import SLURMCluster
cluster = SLURMCluster(
    queue='normal',
    cores=24,
    memory='128GB',
    walltime='02:00:00',
    job_extra=['--constraint=haswell']
)

cluster.scale(jobs=10)  # Request 10 nodes
client = Client(cluster)
```

Dask Cluster Types

Cloud-Native Solutions

Kubernetes Clusters

- Container orchestration for Dask workers
- Auto-scaling based on demand
- Integration with cloud providers (AWS, GCP, Azure)

```python
from dask_kubernetes import KubeCluster
cluster = KubeCluster.from_yaml('worker-spec.yaml')
cluster.scale(20)  # 20 worker pods
client = Client(cluster)
```

Coiled: Managed Dask clusters with auto-scaling and easy deployment.

Saturn Cloud: Provides Dask clusters with Jupyter integration and easy scaling.

AWS Fargate: Serverless Dask clusters on AWS, automatically scaling based on workload.

Leveraging BioHPC for Dask

How to perform a multi-node SlurmRunner job on BioHPC

See ***exercises/session_4/exercise_3.sh***

```bash
#!/bin/bash
#SBATCH --job-name dask-seg # Job name
#SBATCH -p 512GB # Partition name (queue)
#SBATCH -N 6 # Number of nodes
#SBATCH --mem 501760  # Total memory per node (in MB)
#SBATCH -t 2-0:0:00 # Wall time (days-hours:minutes:seconds)
#SBATCH -o /home2/kdean/portal_jobs/job_%j.out
#SBATCH -e /home2/kdean/portal_jobs/job_%j.err
#SBATCH --mail-type ALL
#SBATCH --mail-user kevin.dean@UTSouthwestern.edu

export PATH="/project/bioinformatics/Danuser_lab/Dean/dean/miniconda3/bin:$PATH"
eval "$(/project/bioinformatics/Danuser_lab/Dean/dean/miniconda3/bin/conda shell.bash hook)"
conda --version
source activate dask-nanocourse

```

Leveraging BioHPC for Dask

How to perform a multi-node SlurmRunner job on BioHPC (continued)

See ***exercises/session_4/exercise_3.py***

```python
# Standard Library Imports
import os
import argparse
import logging

# Third Party Imports
import numpy as np
from scipy import ndimage
from dask.distributed import Client
from dask_jobqueue.slurm import SLURMRunner
import dask.array as da

# Parse command line arguments
parser = argparse.ArgumentParser()
parser.add_argument("--job-id", type=str, required=True, help="SLURM Job ID")
args = parser.parse_args()
```

Leveraging BioHPC for Dask

How to perform a multi-node SlurmCluster job on BioHPC

Slurm Clusters

- Use `SLURMCluster` to create a Dask cluster that automatically submits jobs to SLURM.
- Run a single python script or Jupyter notebook that initializes the cluster and submits jobs.
- Cluster configuration directly specified in script, or can be loaded from a YAML file.

Leveraging BioHPC for Dask

How to perform a multi-node SlurmCluster job on BioHPC

- See ***exercises/session_4/exercise_4.ipynb***

```python
# Standard Library Imports
import os
import time
import subprocess

# Third Party Imports
from dask_jobqueue import SLURMCluster
from dask.distributed import Client
from dask import array as da
import numpy as np
from scipy import ndimage

# Figure out what your default `umask` setting is.
result = subprocess.run("umask", shell=True, capture_output=True, text=True)
print("Subprocess umask:", result.stdout.strip())

```

Cluster Management Best Practices

Worker Memory vs. Chunk Size Relationships

- **Match chunk size to worker memory:** Choose chunk sizes small enough that many chunks fit into a worker's RAM concurrently. Partitions larger than a few gigabytes risk exceeding memory. Very tiny partitions (e.g. <1MB) create millions of tasks, overwhelming the scheduler.
- **Use guidelines for chunk sizing:** As a rule of thumb, aim for chunk sizes on the order of 100MB to 1GB for HPC workloads. Chunks much smaller than 100MB often incur high overhead, whereas chunks larger than 1–2GB should only be used if a worker has ample memory per core.
- **Ensure sufficient parallelism:** Having too few chunks can under-utilize the cluster. The number of chunks should be at least equal to (or a multiple of) the total cores in the cluster to keep all workers busy (e.g. ≥2× number of cores) .
- **Tune memory in SLURM jobs:** When using Dask-Jobqueue with SLURM, set each worker's memory limit to match the SLURM allocation (e.g. SLURMCluster(memory="16GB", cores=4)). This ensures Dask's scheduler is aware of real memory per job.

Cluster Management Best Practices

Network Bandwidth Considerations

- **Leverage high-speed interconnects:** On HPC clusters with InfiniBand or similar, instruct Dask to use the high-bandwidth interface. For example, pass --interface ib0 to dask-scheduler/dask-worker or interface='ib0' in your SLURMCluster so Dask uses InfiniBand instead of Ethernet .
- **Minimize data transfer volume:** Avoid repeatedly shipping large datasets through the scheduler or client. Instead, load data in parallel on the workers or use methods like client.scatter to distribute data once upfront. This reduces network load and prevents the scheduler from becoming a bottleneck due to large data handling.
- **Be cautious with shuffles and broadcasts:** Operations like Dask DataFrame shuffles or broadcasting variables to all tasks can saturate network bandwidth. If possible, repartition or filter data to reduce shuffle sizes, or use algorithms that localize data.
- **Monitor communication in the dashboard:** The Task Stream plot highlights communication delays (e.g. red segments indicate waiting on data from peers).

Cluster Management Best Practices

Storage Locality Optimization

- **Avoid spilling to shared filesystems:** By default, Dask workers spill excess data to the temp directory, which on HPC systems is often a shared network filesystem (NFS, Lustre, etc.). It's best to minimize reliance on a shared FS for intermediate data.
- **Use node-local storage for temp data:** If compute nodes have local SSD scratch space, direct Dask to use them for spilling and scratch files. For example, launch workers with `--local-directory /local/scratch` or configure `local_directory=...` in the cluster setup.
- **Adjust spilling behavior if needed:** On systems without any local storage, you may choose to disable disk spilling to avoid pounding the shared filesystem. Dask allows this via config: e.g., setting `distributed.worker.memory.spill`: false (and memory.target: false) will prevent spilling and instead pause tasks when memory is full . In such cases, it's crucial to fit working data in memory or increase memory per worker.
- **Persist data to improve locality:** If a dataset will be reused across multiple computations, call `persist()` on it to keep it in distributed memory (or explicitly write to a node-local store).

Cluster Management Best Practices

Dask Dashboard for Real-Time Monitoring

- **Live cluster insight:** The Dask dashboard (a web UI on the scheduler, default port 8787) provides real-time visualization of your computations. It shows tasks progressing, cluster CPU utilization, memory usage per worker, network throughput, etc., helping you understand the state of your cluster at a glance .
- **Identify bottlenecks visually:** Use the dashboard's plots to catch performance issues. For example, red for waiting on data, orange for disk I/O, etc.
- **Build parallel intuition:** The dashboard is an excellent teacher of distributed performance; visual feedback helps develop intuition on how tasks, data, and resources interact in Dask.
- **Accessing the dashboard on HPC:** In a SLURM cluster environment, the dashboard is typically not directly visible on your local machine. You can forward the port via SSH to view it in a browser.

```
`ssh -N -L 44460:localhost:44460 your-cluster-login@nucleus.biohpc.swmed.edu`
#You can then access this at `http://localhost:44460/status`
```

- **Dashboard panels and usage:** Explore different tabs like Status, Workers, Task Graph, Profile, etc. The Workers tab shows per-worker CPU and memory; the Graph tab visualizes the task dependency graph (useful to spot stragglers or uneven task distribution).

Cluster Management Best Practices

Performance Profiling Tools

- **Programmatic profiling with `Client.profile()`:** In addition to the live view, you can capture profiling data in your code. The `Client.profile()` method returns the collected profile info, which you can save to disk (e.g., `client.profile(filename=\"dask-profile.html\")`). This HTML profile can be viewed later, allowing offline analysis of a run's performance characteristics.
- **Performance reports:** For a comprehensive performance snapshot, use `distributed.performance_report`. Wrapping your computations in with `performance_report(filename=\"dask-report.html\")`: will produce an HTML report containing major diagnostics – task timeline, worker profiles, transfer statistics, etc. This is useful for sharing results of a performance test or debugging session with colleagues, or analyzing performance post hoc.
- **Classic profilers vs Dask:** Traditional Python profilers (e.g. cProfile) don't capture multi-threaded or multi-process workloads well. Dask's built-in tools are designed for distributed execution. If needed, you can still profile a portion of your code in isolation (e.g., run a representative task function locally under cProfile).

Adapting Code for Distributed Dask

Lab 3: 3:30 – 4:15 PM

Spend some time adapting your existing code to run with Dask.

Pair up, have fun, see if you can launch a multi-node Dask job on BioHPC, and share your results with the class.