Smallest font

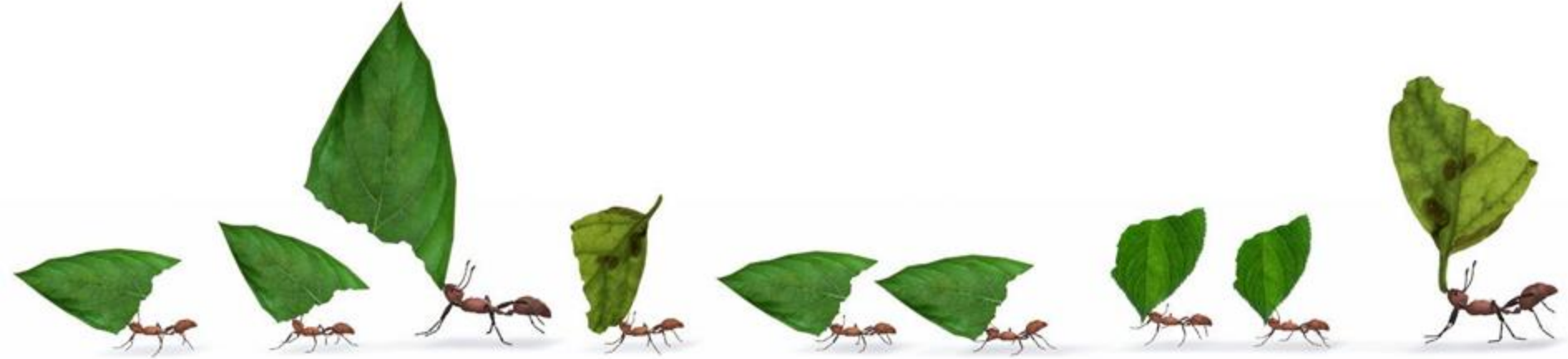
Please turn off and put away your cell phone

Calibration slide

These slides are meant to help with note-taking
They are no substitute for lecture attendance

Smallest font

Big Data



NYU | Center for Data Science

Week 08: Dask

DS-GA 1004: Big Data

Announcements

- HW4 (Spark) will be released within the next couple of days
- Outstanding grades will also be released this week

The story so far…

How do we store and process large collections of data?

- **Collections of files**
  - Unstructured
  - Lots of custom coding
  - Very flexible
  - Parallelism? *You're on your own…*
- **Relational databases**
  - Restricted / structured (tabular) data
  - Standard interface (SQL)
  - Somewhat flexible
  - Parallelism? *It's complicated…*
- **Map-Reduce + HDFS**
  - Data is less structured than RDBMS
  - Restricted coding interface (map/reduce)
  - Very parallel!
- **Spark**
  - Structured data like RDBMS
  - Distributed storage (HDFS)
  - Standard-ish interface (SQL or Spark object API)
  - Very parallel!

Spark is very useful for many things!

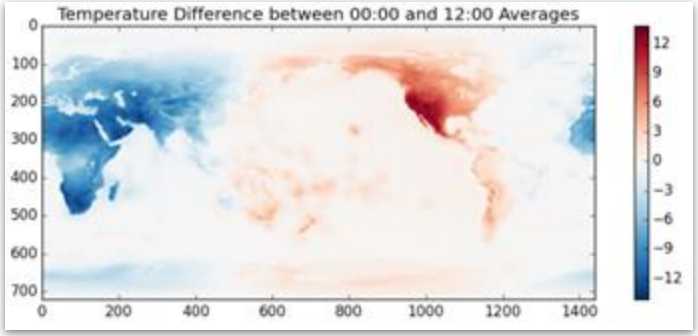
- Spark integrates nicely with Java-based tools (**Hadoop ecosystem**)
  - HDFS, Parquet, YARN scheduler, etc…
- Spark is great for DataFrames and SQL-like processing
  - *Graphs also, though we haven't gotten to that yet – we'll see this later in the course*
- >10 years old now, **implementation is mature and stable**
- After RDBMS/SQL, **probably the most widely used software** we cover in this course

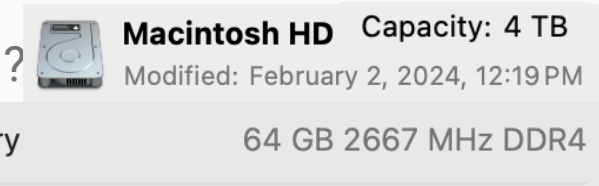
But Spark is not useful for everything…

- Can you think of a use case where Spark would be unsuitable or a computation that would be difficult to do with Spark?

Some things are still difficult with Spark…

- Data that doesn't naturally fit the RDD/DataFrame model:
- Integration with scientific Python stack
- (Modern) Machine learning tools:
  - sklearn, pytorch, etc…
- Scaling **down** vs. scaling **out**:
  - What if the data fits into laptop storage, but not RAM?
  - Do you really need a cluster?





Dask: An open source parallel computing library

[Rocklin, 2015]

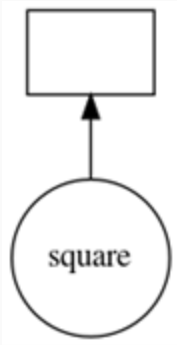
- Python-based distributed computation
- Many common design principles with **Spark**
  - Computation graphs (**lineage graphs**)
  - Delayed computation (**RDD/transformations**)
  - Collections-based interfaces (**DataFrames**)
- Some key differences from Spark:
  - **Prioritizes array-based (numpy-like) computation**
  - **Designed to support single-machine, out-of-core use**

"**Out-of-core computation**": Processing data that exceeds the size of main memory of the computer

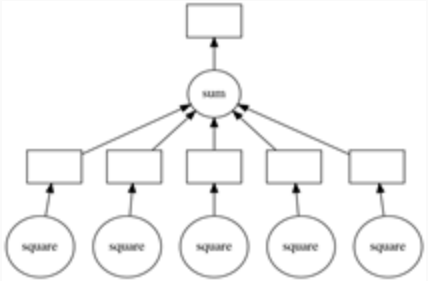
Delayed computation and task graphs in Dask

- Dask builds complex computations by composing deferred computations into a **task graph.** Like in Spark, nothing happens until you take an **action**.

```
import dask

def square(x):
        return x**2

f = dask.delayed(square)
y = f(5)
y.visualize()  # draw computation graph
```




```
g = dask.delayed(sum)

z = g([f(x) for x in range(5)])

z.visualize()

z.compute()
```



There are several types of collections in Dask

- Bags
  - Distributed collections of arbitrarily structured data
  - **Most similar to:** RDD
- DataFrames
  - Distributed collection of structured, tabular data
  - **Most similar to:** Spark DataFrame (but built on pandas instead of RDDs)
- Arrays
  - Distributed n-dimensional arrays
  - **Most similar:** to numpy.ndarray (but distributed!)

Collections interfaces: **Bags**

- Dask **bags** are loosely analogous to Spark RDDs
  - But you can think of it more like a (parallel) python list
- Unordered collection of generic Python objects
  - Partitions into subsets (sub-bags) to parallelize
- Implements some basic operations
  - map, filter, join, sum, etc.

```
import dask.bag as db

b = db.from_sequence(range(5))

c = b.map(square)

c.compute()   # [0, 1, 4, 9, 16]

c.sum().compute()  # 30
```

- A good choice for initial processing and handling of structured objects
  - If your data is tabular or array-based, there are better choices than bags

Dask Bags vs. Spark RDDs: Similarities and differences

- Both allow to partition a collection across multiple machines / cores
- Both are immutable
- RDDs have **types** (e.g. `RDD[Integer]`)
- Bags are **untyped**, contents can be mixed

Tips for working with bags

- A common workflow is:

  Raw data → Bags → DataFrame → (more sophisticated analysis)

- The earlier you can reduce the size of your data, the better!
  - Less data moving through the system is a good thing!
- Recommended: **maps** and **filters** on bags over DataFrame operations when simplifying data

Tips for working with bags

- Common workflow is:

  Raw data → Bags → DataFrame → (more sophisticated analysis)

- The earlier you can reduce the size of your data, the better!
  - Less data moving through the system is a good thing!
- Prefer **maps** and **filters** on bags over DataFrame manipulations when simplifying data
- **HOWEVER**, bag operations are generally slower than DataFrame operations
  - For the same reason that pandas/numpy is faster than vanilla python code
- Because bags have so few restrictions, Dask can't assume much.
  - You'll have to think carefully about optimizing your code.

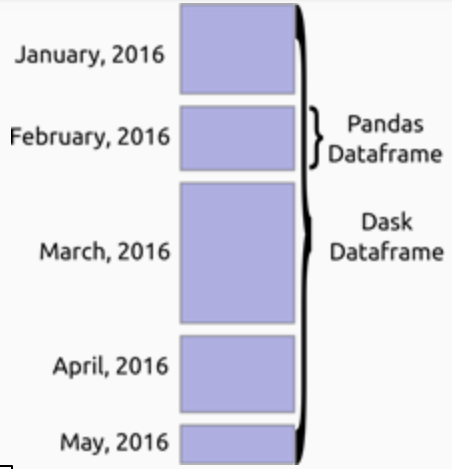
Another practical tip: Bag folding vs grouping

```
import dask.bag as db

b = db.from_sequence(range(10))
iseven = lambda x: x % 2 == 0
add = lambda x, y: x + y

dict(b.foldby(iseven, add))
```

- Try to **avoid using groupBy** on bags
  - This requires much inter-worker communication, and is slow!
  - This is the “wide dependency” problem from Spark again
- Use **fold**/**foldBy** if possible
  - Similar benefits to a **combiner** in map-reduce
  - Perform local aggregation (within partition) first to reduce shuffling
  - Same restrictions apply (associative, commutative)
- You supply a key function and a binary operation

Collections interfaces: **DataFrames**

- Just like you'd expect, similar to Spark DataFrames
  - Uses Pandas internally, interface is basically the same
- Parallelism (partitioning) is over subsets of **rows.** Each partition is a Pandas dataframe
- Good choice for data that can naturally split into multiple CSV files (or Parquet partitions)




```
import dask.dataframe as dd

df = dd.read_csv('*.csv')
df.mean().compute()
```

Example image from https://docs.dask.org/en/latest/dataframe.html

Dask DataFrames and partition management

- **Managing your partitions is important!**
  - Think about how your data changes through the computation graph
- If you're **filtering out records**, you may end up with many (near) empty partitions
- Try to keep your partitions **full and balanced**
- Expect to work harder than in Spark for good performance

```
df = dd.read_csv('s3://bucket/path/to/*.csv')
df = df[df.name == 'Alice']  # only 1/100th of the data
df = df.repartition(npartitions=df.npartitions // 100)
df = df.persist()  # if on a distributed system
```

https://docs.dask.org/en/stable/dataframe-best-practices.html

Aside

Here’s a real-life example how one could use Dask Dataframes

Example: machine listening evaluation

- Say you have 10 models for automatic audio segmentation that you want to compare
- You have a dataset of 2000 audio recordings
  ⇒ **20,000 model outputs** to evaluate
- (Pre-computed) **model outputs** are, for each recording, a sequence of labeled time intervals:

```
# start_time,     end_time,     label
0,                5,            silence
5,                23.2,         intro
23.2,             44.9,         verse
…
```

- Model evaluator compares **reference annotation** to **estimate annotation** for one track, and produces a dictionary of scores along different metrics.
- What you want: a DataFrame containing:
  model id, recording id, [scores for each metric]

Example: machine listening evaluation

- Solution:
  - Store model all outputs on disk as “{model_id}/{recording_id}.txt”

```
files = glob('*/*.txt')
```

  - Create a delayed function to map filename to scores (calls the evaluator)

```
my_evaluator(file_name):
    # load estimate and reference data
    # call the evaluator
    return scores_dict
evaluator = delayed(my_evaluator)
```

  - Create a bag from the delayed function

```
results = db.from_delayed([evaluator(estimate) for estimate in files])
```

  - Convert the bag to a dataframe and save

```
results_df = results.to_dataframe(...)  # not pictured: schema definition
results_df.to_parquet('output.parquet')
```

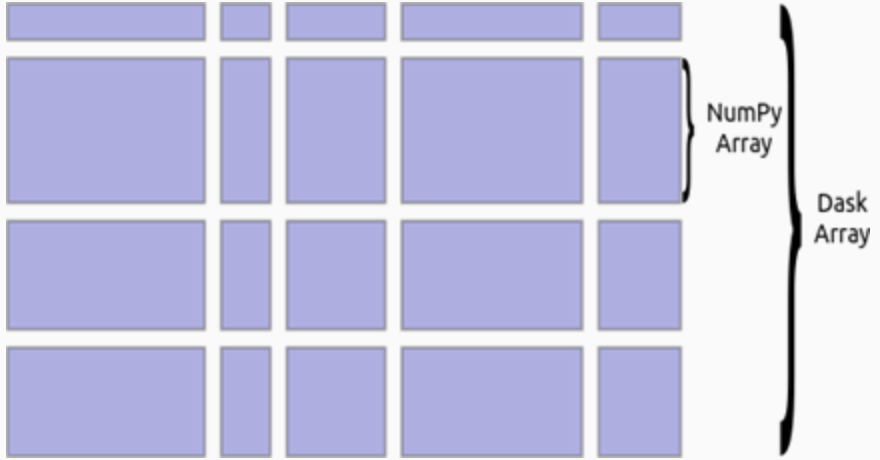
What did this save us?

- Everything is done with basic python functions
  - **Core computation (evaluator) was not written explicitly for Dask**
- The problem itself is “embarrassingly parallel”
  - aka “all map, no reduce”
  - Don’t need to think too hard about partition structure
- Coding this up with Dask is about as simple as it gets
  - Probably could also be done with MrJob, but with a lot more overhead

Back to other Dask collections interfaces…

Collections interfaces: **Arrays**

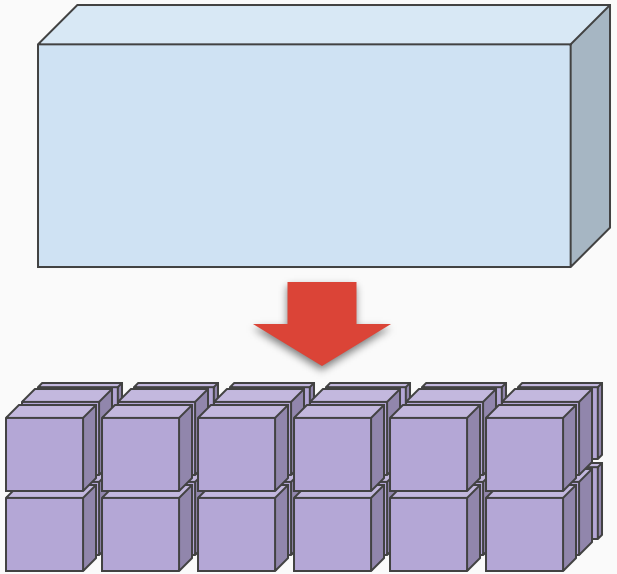
- Dask Arrays work like NumPy arrays
- **Parallelism is not limited to rows**
  - You can define **chunks** along each dimension
- Large arrays are assembled implicitly from many small arrays
- Most* numpy operations work automatically



Example image from https://docs.dask.org/en/latest/array.html

Array chunking example

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

# Make some random noise: 2000 x 6000 x 5
x = np.random.randn(2000, 6000, 5)

# Slice it up into chunks
x_parallel = da.from_array(x, chunks=(1000, 1000, 2))
```



Scaling down: why a single machine?

- **Many “big data” jobs do not really need a cluster!**
  - Data fits on a hard drive, but not in RAM (“core memory”)
  - NumPy (& friends) generally assume fully observed, in-memory data
- In this case, working on small chunks at a time is sufficient
  - Coding this by hand can be tedious / error-prone
- Dask simplifies this, and makes it easy to migrate to a cluster if necessary

Numpy vs. Dask arrays: Simplicity vs. scalability

- If the problem fits into memory (data is not prohibitively large), numpy outperforms Dask, due to lack of overhead (scheduling, managing threads, etc.)
- If the data exceeds system memory, a dask array might be more suitable. Under the hood, a dask array consists of chunks, each of which is a numpy array.
- Processing of these chunks with Dask can be parallelized, both on a single machine and on a cluster (more on this on the next slide).
- Numpy is not inherently multi-threaded, whereas Dask is (on a multi-core system, Dask can often yield near-linear speedups, if tasks are embarrassingly parallel).

Schedulers: how does Dask code run?

- On a single machine, you have several options:
  - Parallelizing into multiple **processes**
  - Parallelizing into multiple **threads**
  - Both
- Or you can run on a cluster
  - Execution can look very similar to Spark (eg YARN jobs)
  - Data is transferred automatically / as needed

Processes

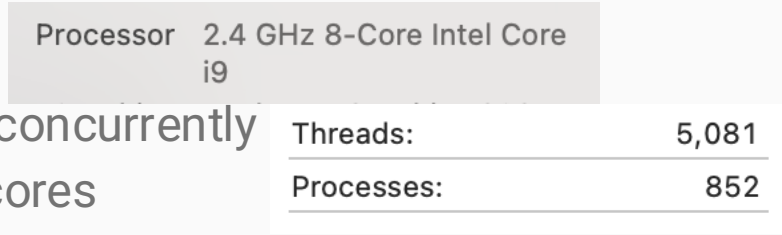
- An instance of a program with a self-contained execution environment with own resources
- Shared data must be sent between processes
- Processes do not block each other

Threads

- Components of a single python process
- Shared memory between threads
- Certain python operations can block computation for all threads

Multi-thread vs. multi-process parallelization

- Modern machines often have multiple cores.
- There are many processed and threads running concurrently
- One thread runs on one core, not split between cores
- However, each core can only execute one thread at a time, putting a limit on true parallelization (threads are competing with each other.
- If that is a concern, cloning the process and distributing them over multiple CPU cores can yield true parallelization.



Using Dask with HDF5 (not the same as HDFS!)
HDF5 = Hierarchical data format, version 5

- Basically a file-system within a file
  - (Hierarchical) Directory structures
- In python, use the **h5py** package
- HDF5 supports memory-mapped file access (non-human readable binaries).
- Data is memory-mapped, not loaded

```
import h5py

data = h5py.File('myfile.h5', mode='r')

x = data['/x']
y = data['/y']
z = data['/path/to/z']
```

```
import dask.array as da

x_parallel = da.from_array(x, chunks=(1000, 1000))
```

Does Dask replace Spark?

- It depends…
  - https://docs.dask.org/en/latest/spark.html summarizes use-cases and differences
- Pros for Dask:
  - Do you need to integrate with the SciPy stack? (Matplotlib, sklearn, etc)
  - Do you need to work with dense / multi-dimensional data?
  - Custom algorithms / advanced machine learning? GPUs?
- Pros for Spark:
  - More mature, possibly more stable / safe
  - More “high-level” -- you don’t need to think as much about the compute graph
  - Probably faster / better optimized for DataFrame crunching
  - Better support for large graph data

HPC: Dataproc and Greene

- So far, we've been using a Hadoop cluster (**Dataproc**)
  - HDFS storage
  - MapReduce + Spark jobs (YARN)
- We also have the **Greene** cluster for less restrictive computation
  - Network-accessible file storage (IBM GPFS, **not** HDFS)
  - Traditionally preferred if you have "embarrassing parallelism", but Dask can run on it too

Dask is not the only parallelization option

- There is an increasing number of parallel versions of classic Python libraries.
- For instance: Polars, a parallel version of Pandas.
- Polars is implemented in Rust, to give C-like speed, but exposed to Python.
- In contrast to Pandas (singe-core), Polars runs on all CPU cores.
- Polars: Column oriented storage (via Apache Arrow)
- Polars: Lazy evaluation (optimized query plans)
- Polars: Inherent multi-threading
- Pandas dataframes: Mutable. Polars: Immutable
- …

Next time (next week: Spring break)

- Starting applications
- Similarity/search