Dask data processing

Mirko Mariotti
mirko.mariotti@unipg.it

original slides by Tommaso Tedeschi
tommaso.tedeschi@pg.infn.it

ICSC
Centro Nazionale di Ricerca in HPC,
Big Data and Quantum Computing

INFN
PERUGIA
Istituto Nazionale di Fisica Nucleare
Sezione di Perugia

The problem

- The majority of **small** data science problems can be tackled with a Numpy/Pandas/Custom code combination:
  - In particular, pandas is great for tabular datasets that fit in memory. A general rule of thumb for pandas is: “Have 5 to 10 times as much RAM as the size of your dataset” [Wes McKinney (2017) in “10 things I hate about pandas”]
- What if we have to tackle **big** data problems? This implies two things:
  - Datasets to analyze will be (much) larger that your memory
  - We would like to use all available computing power (on my machine or on many different machines)
  - HERE IS WHEN DASK COMES TO THE RESCUE!

What is Dask

Dask is a parallel and distributed computing library that scales the existing Python and PyData ecosystem.

Dask can scale up to your full laptop capacity and out to a cloud cluster.

Dask provides multi-core and distributed+parallel execution on larger-than-memory datasets

Main virtues

- **Familiar**:
  - Provides parallelized NumPy array and Pandas DataFrame objects
- **Flexible**:
  - Provides a task scheduling interface for more custom workloads and integration with other projects.
- **Native**:
  - Enables distributed computing in pure Python with access to the PyData stack.
- **Fast**:
  - Operates with low overhead, low latency, and minimal serialization necessary for fast numerical algorithms
- **Scales up**:
  - Runs resiliently on clusters with 1000s of cores
- **Scales down**:
  - Trivial to set up and run on a laptop in a single process
- **Responsive**:
  - Designed with interactive computing in mind, it provides rapid feedback and diagnostics to aid humans

Main concepts

Main concepts behind Dask**:**

- **Parallelism**: Uses all of the cores on your computer
- **Larger-than-memory**: Lets you work on datasets that are larger than your available memory by breaking up your array into many small pieces, operating on those pieces in an order that minimizes the memory footprint of your computation, and effectively streaming data from disk.
- **Laziness:** Most Dask interfaces are lazy, meaning that they do not evaluate until you explicitly ask for a result
- **Blocked Algorithms**: Perform large computations by performing many smaller computations.

Main components

Two main pillars:

- **Dynamic task scheduling** optimized for interactive computational workloads

- **“Big Data” collections** like parallel arrays, dataframes, and lists that extend common interfaces like NumPy, Pandas, or Python iterators to larger-than-memory or distributed environments. These parallel collections run on top of dynamic task schedulers.

Data structures

- **High-level collections**: Dask provides high-level Array, Bag, and DataFrame collections that mimic NumPy, lists, and pandas but can operate in parallel on datasets that don’t fit into memory.

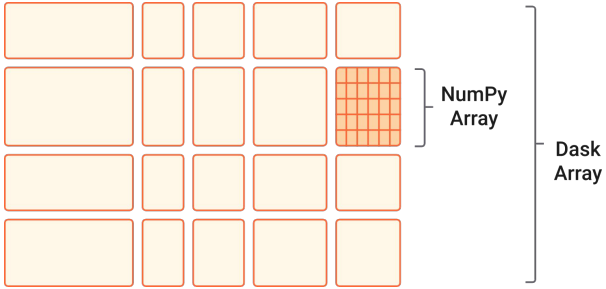
- **Low-level collections**: Dask also provides low-level Delayed and Futures collections that give you finer control to build custom parallel and distributed computations.

Dask Arrays

A Dask array is composed of many Numpy arrays (or “duck arrays” that are sufficiently NumPy-like in API), arranged into chunks within a grid

- This lets us compute on arrays larger than memory using all of our cores. We coordinate these blocked algorithms using Dask graphs

Dask arrays support a large subset of the Numpy API (Dask Array also implements a subset of the scipy.stats package)



- Arithmetic and scalar mathematics: `+, *, exp, log, ...`
- Reductions along axes: `sum(), mean(), std(), sum(axis=0), ...`
- Tensor contractions / dot products / matrix multiply: `tensordot`
- Axis reordering / transpose: `transpose`
- Slicing: `x[:100, 500:100:-2]`
- Fancy indexing along single axes with lists or NumPy arrays: `x[:, [10, 1, 5]]`
- Array protocols like `__array__` and `__array_ufunc__`
- Some linear algebra: `svd, qr, solve, solve_triangular, lstsq`
- ...

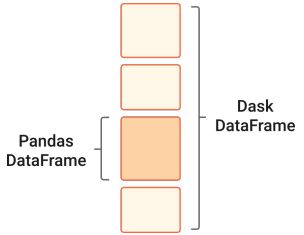
https://docs.dask.org/en/latest/array.html

Dask Dataframes

One Dask DataFrame is comprised of many in-memory pandas DataFrames separated along the index

- these pandas objects may live on disk or on other machines.
- one operation on a Dask DataFrame triggers many pandas operations on the constituent pandas DataFrames in a way that is mindful of potential parallelism and memory constraints
- It is used in situations where pandas is commonly needed, usually when pandas fails due to data size or computation speed:
  - Manipulating large datasets, even when those datasets don't fit in memory
  - Accelerating long computations by using many cores
  - Distributed computing on large datasets with standard pandas operations like groupby, join, and time series computations

They support a large subset of the Pandas API



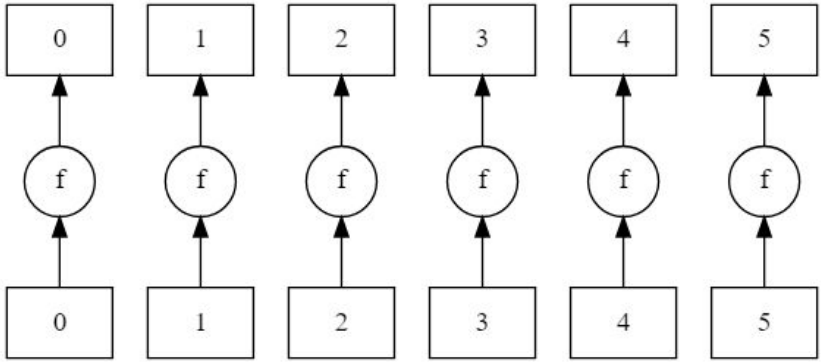
- Element-wise operations: `df.x + df.y`, `df * df`
- Row-wise selections: `df[df.x > 0]`
- Loc: `df.loc[4.0:10.5]`
- Common aggregations: `df.x.max()`, `df.max()`
- Is in: `df[df.x.isin([1, 2, 3])]`
- Date time/string accessors: `df.timestamp.month`
- groupby-aggregate (with common aggregations): `df.groupby(df.x).y.max()`, `df.groupby('x').min()` (see Aggregate)
- groupby-apply on index: `df.groupby(['idx', 'x']).apply(myfunc)`, where `idx` is the index level name
- value_counts: `df.x.value_counts()`
- Drop duplicates: `df.x.drop_duplicates()`
- Join on index: `dd.merge(df1, df2, left_index=True, right_index=True)` or `dd.merge(df1, df2, on=['idx', 'x'])` where `idx` is the index name for both `df1` and `df2`
- Join with pandas DataFrames: `dd.merge(df1, df2, on='id')`
- Element-wise operations with different partitions / divisions: `df1.x + df2.y`
- Date time resampling: `df.resample(...)`
- Rolling averages: `df.rolling(...)`
- Pearson's correlation: `df[['col1', 'col2']].corr()`
- Set index: `df.set_index(df.x)`
- groupby-apply not on index (with anything): `df.groupby(df.x).apply(myfunc)`
- Join not on the index: `dd.merge(df1, df2, on='name')`

https://docs.dask.org/en/latest/dataframe.html

Dask Bags

Dask Bags coordinate many Python lists or Iterators, each of which forms a partition of a larger collection

- Implements operations like `map`, `filter`, `groupby` and aggregations on collections of Python objects
- Dask bags are often used to parallelize simple computations on unstructured or semi-structured data like text data, log files, JSON records, or user defined Python objects.



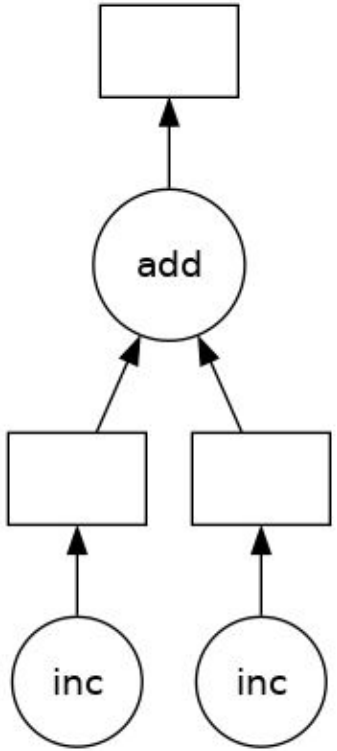
https://docs.dask.org/en/stable/bag.html

Dask Delayed

The **Dask delayed** function decorates your Python functions so that they operate lazily.

- Rather than executing your function immediately, it will defer execution, placing the function and its arguments into a task graph with dependencies
- Useful when your problem doesn't fit into one of the higher-level collections

Sometimes you want to create and destroy work during execution, launch tasks from other tasks, etc. For this, see the Futures interface.



https://docs.dask.org/en/latest/delayed.html

Dask Futures

Submit arbitrary functions for computation in a parallelized, eager, and non-blocking way

- This interface is good for arbitrary task scheduling like dask.delayed, but is immediate rather than lazy
  - more flexibility in situations where the computations may evolve over time.
- The intermediate results, represented by futures can be passed to new tasks without having to pull data locally from the cluster:
  - new operations can be setup to work on the output of previous jobs that haven't even begun yet.

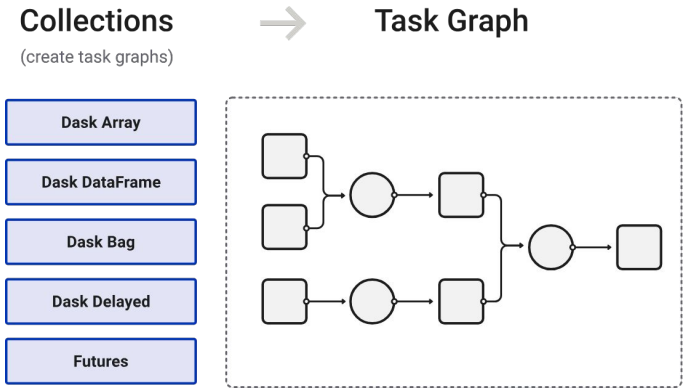
Only available when working with distributed schedulers! (see next slides

https://docs.dask.org/en/latest/futures.html

From functions to graphs

Dask collections and the fine-grained APIs generate **task graphs:**

- Each node in the graph is a normal Python function
- edges between nodes are normal Python objects that are created by one task as outputs and used as inputs in another task
- Internally, Dask encodes algorithms in a simple format involving Python dicts, tuples, and functions
- Dask needs to execute graphs on parallel hardware:
  - This is the job of a task scheduler

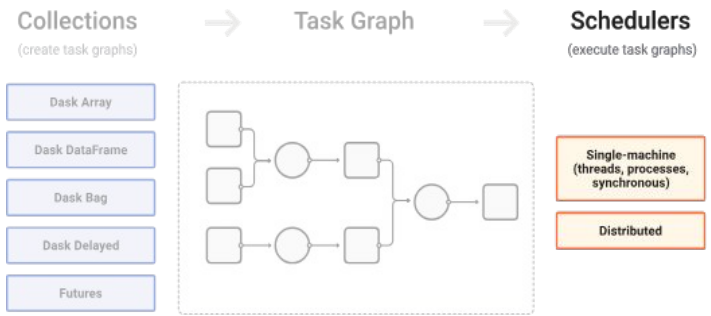


Scheduling

Different task schedulers exist, and each will consume a task graph and compute the same result, but with different performance characteristics. Two families:

- **Single-machine scheduler**: This scheduler provides basic features on a local process or thread pool.
  - It was made first and is the default and is simple and cheap to use, although it can only be used on a single machine and does not scale

- **Distributed scheduler**: This scheduler is more sophisticated, offers more features, but also requires a bit more effort to set up.
  - locally or distributed across a cluster
  - more on this on other lectures



https://docs.dask.org/en/stable/scheduling.html

Single-machine schedulers

- **Single-threaded synchronous scheduler**
  - executes all computations in the local thread with no parallelism at all. This is particularly valuable for debugging and profiling
  - `dask.config.set(scheduler='synchronous')`
- **Threaded scheduler**
  - executes computations with a local concurrent.futures.ThreadPoolExecutor.
  - It is lightweight and requires no setup.
  - It introduces very little task overhead (around 50us per task) and, because everything occurs in the same process, it incurs no costs to transfer data between tasks.
  - `dask.config.set(scheduler='threads')`
- **Multiprocessing scheduler**
  - executes computations with a local concurrent.futures.ProcessPoolExecutor.
  - It is lightweight to use and requires no setup.
  - Every task and all of its dependencies are shipped to a local process, executed, and then their result is shipped back to the main process.
  - `dask.config.set(scheduler='processes')`

Single-machine schedulers

Why there are two (concurrent) single-machine scheduler ?

- `dask.config.set(scheduler='threads')`
- `dask.config.set(scheduler='processes')`

Python uses the global interpreter lock (GIL): only one thread (per process) can allocate memory and reference counting.

- Threads are not so performant in pure python code as you may think. They may be if the code just wrap something the handle threads properly. For example the pandas functions groupby, nsmallest, value_counts.
- On the other hand, processes can fill your cores fully, but they are not as cheap as the threads and you will introduce a lot of overhead.

When talking performances you have to know your hardware, your application and make the right compromise

Collections scheduler defaults

The **dask collections each have a default scheduler**:

- `dask.delayed, dask.array` and `dask.dataframe` use the threaded scheduler by default
- `dask.bag` uses the multiprocessing scheduler by default.

For most cases, the default settings are good choices. However, sometimes you may want to use a different scheduler. if your computation is dominated by processing pure Python objects like strings, dicts, or lists, then you may want to try one of the process-based schedulers

General best practices

- Start Small:
  - Parallelism brings extra complexity and overhead
  - Before adding a parallel computing system try some alternatives: better algorithms or file formats, compiled code..
- Avoid Very Large Partitions:
  - Your chunks of data should be small enough so that many of them fit in a worker's available memory at once
  - Dask will likely manipulate as many chunks in parallel on one machine as you have cores on that machine.
  - it's common for Dask to have 2-3 times as many chunks available to work on so that it always has something to work on.
- Avoid Very Large Graphs:
  - Every task comes with some overhead: this is somewhere between 200us and 1ms

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

Best practices - Arrays

- If your data fits comfortably in RAM and you are not performance bound, then using NumPy might be the right choice
  - Dask adds another layer of complexity which may get in the way
  - If you are just looking for speedups rather than scalability then you may want to consider a project like Numba
- Select good chunk size:
  - While optimal sizes and shapes are highly problem specific
  - it is rare to see chunk sizes below 100 MB in size.
  - If you are dealing with float64 data then this is around (4000, 4000) in size for a 2D array or (100, 400, 400) for a 3D array.
- When reading data you should align your chunks with your storage format.
  - Most array storage formats store data in chunks themselves

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

Best practices - Dataframes

- If you can, use Pandas:
  - For data that fits into RAM, pandas can often be faster and easier to use than Dask DataFrame
  - When you've reduced things to a more manageable level, persist and switch to Pandas
- Use the Index, Avoid Full-Data Shuffling as much as you can and persist intelligently (when running distributed):
  - Dask DataFrame can be optionally sorted along a single index column
    - some operations against the index column can be very fast
  - Setting an index is an important but expensive operation: do it infrequently and persist afterwards
    - It is often ideal to load, filter, and shuffle data (to set to set an intelligent index) once and keep this result in memory
- Repartition to Reduce Overhead:
  - Partitions should fit comfortably in memory (smaller than a gigabyte) but also not be too many
  - After filters, it is wise to regroup your many small partitions into a few larger ones

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

Best practices - Delayed

- Call delayed on the function, not the result
  - compute on lots of computations at once
- Don’t mutate inputs:
  - If you need to use a mutable operation, then make a copy within your function first
- Avoid global state:
  - Using global state might work if you only use threads
- Don’t call dask.delayed on other Dask collections or within delayed functions:
  - When you place a Dask array/DataFrame into a delayed call, that function will receive the NumPy or Pandas equivalent
  - this might crash your workers
- Break up computations into many pieces:
  - You achieve parallelism by having many delayed calls, not by using only a single one
- Avoid too many tasks:
  - Every delayed task has an overhead of a few hundred microseconds
  - break up your many tasks into batches or use one of the Dask collections to help you

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

Dask ecosystem

In addition to the core Dask library and its distributed scheduler, the Dask ecosystem connects several additional initiatives, including:

- Dask-ML (parallel scikit-learn-style API)
- Dask-image
- Dask-cuDF
- Dask-sql
- Dask-snowflake
- Dask-mongo
- Dask-bigquery

Community libraries that have built-in dask integrations like:

- Xarray
- XGBoost
- Prefect
- Airflow

Dask deployment libraries:

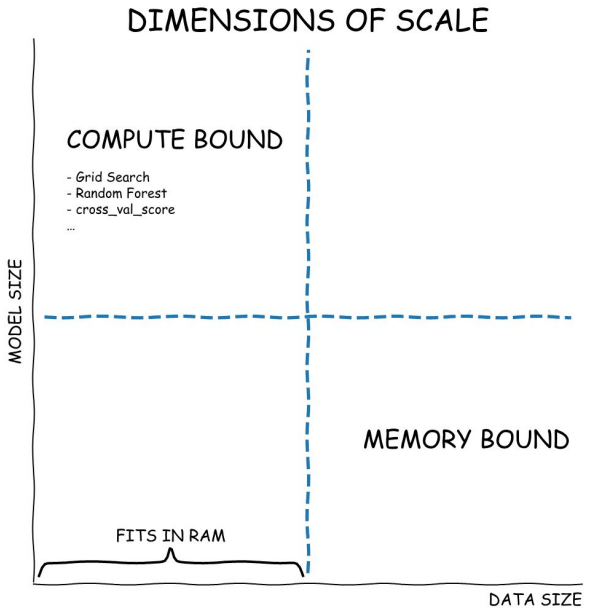
- Dask-kubernetes
- Dask-YARN
- Dask-gateway
- Dask-cloudprovider
- Dask-jobqueue

Dask-ML

Dask-ML provides scalable machine learning in Python using Dask alongside popular machine learning libraries like Scikit-Learn (+Joblib), XGBoost, LightGBM, PyTorch (via Skorch) and Keras (via SciKeras) in particular for:

- Scaling Model Size:
  - Dask's joblib backend to parallelize Scikit-Learn directly
  - hyper-parameter optimizers
- Scaling Data Size:
  - Dask's high-level collections like combined with one of Dask-ML's estimators that are designed to work with Dask collections

Dask-ML endeavors to provide a single unified interface around the familiar NumPy, Pandas, and Scikit-Learn APIs



https://ml.dask.org/

When to choose Dask for distr comp?

- When you prefer Python or native code, or have large legacy code bases that you do not want to entirely rewrite
- When you want a lighter-weight transition from local computing to cluster computing
- When you want to interoperate with other technologies and don’t mind installing multiple packages

Let’s see some examples

https://github.com/SOSC-School/SOSC24-livesessions/tree/main/day2/Dask