UMD DATA605 - Big Data Systems

Python Dask

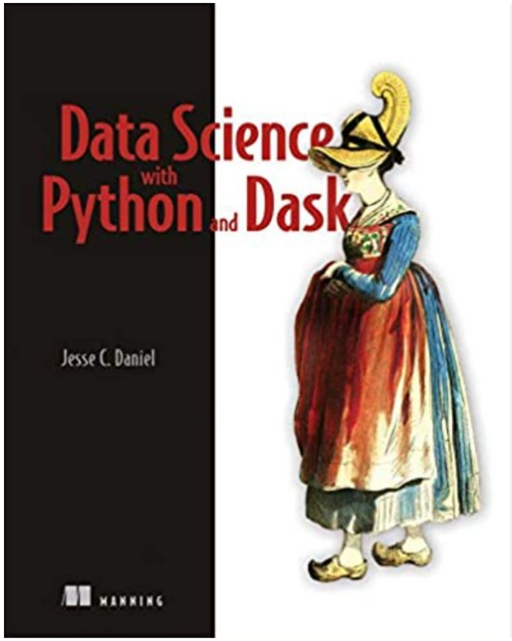
GP Saggese

gsaggese@umd.edu

with thanks to Alan Sussman, Amol Deshpande, and authors of www.mmds.org

Dask: Resources

- Web resources:
  - Dask project
  - Dask examples
- Tutorial
  - Dask_tutorial
  - Dask_advanced_tutorial
- Class project
- Mastery
  - Data science with Python and Dask, 2019
    - Amazon



Dataset Size Issues

- **Small datasets**
  - < 1 GB
  - Fits into RAM
  - Manipulation doesn't require paging to disk
- **Medium dataset**
  - < 1TB
  - Doesn't fit into RAM
  - Fits into local disk
    - Performance penalty imposed by using local disk
  - Need multiple CPU cores
    - Difficult to take advantage of parallelism with Python / Pandas
- **Large dataset**
  - > 1TB
  - Doesn't fit into RAM
  - Doesn't fit into local disk
  - Need multiple servers
    - Python / Pandas were not built to operate on distributed datasets
    - Use frameworks for massive datasets
    - E.g., Hadoop, Spark, Dask, Ray

	outlook	temp	humidity	windy	play
0	sunny	hot	high	False	no
1	sunny	hot	high	True	no
2	overcast	hot	high	False	yes
3	rainy	mild	high	False	yes
4	rainy	cool	normal	False	yes
5	rainy	cool	normal	True	no
6	overcast	cool	normal	True	yes
7	sunny	mild	high	False	no
8	sunny	cool	normal	False	yes
9	rainy	mild	normal	False	yes
10	sunny	mild	normal	True	yes
11	overcast	mild	high	True	yes
12	overcast	hot	normal	False	yes
13	rainy	mild	high	True	no

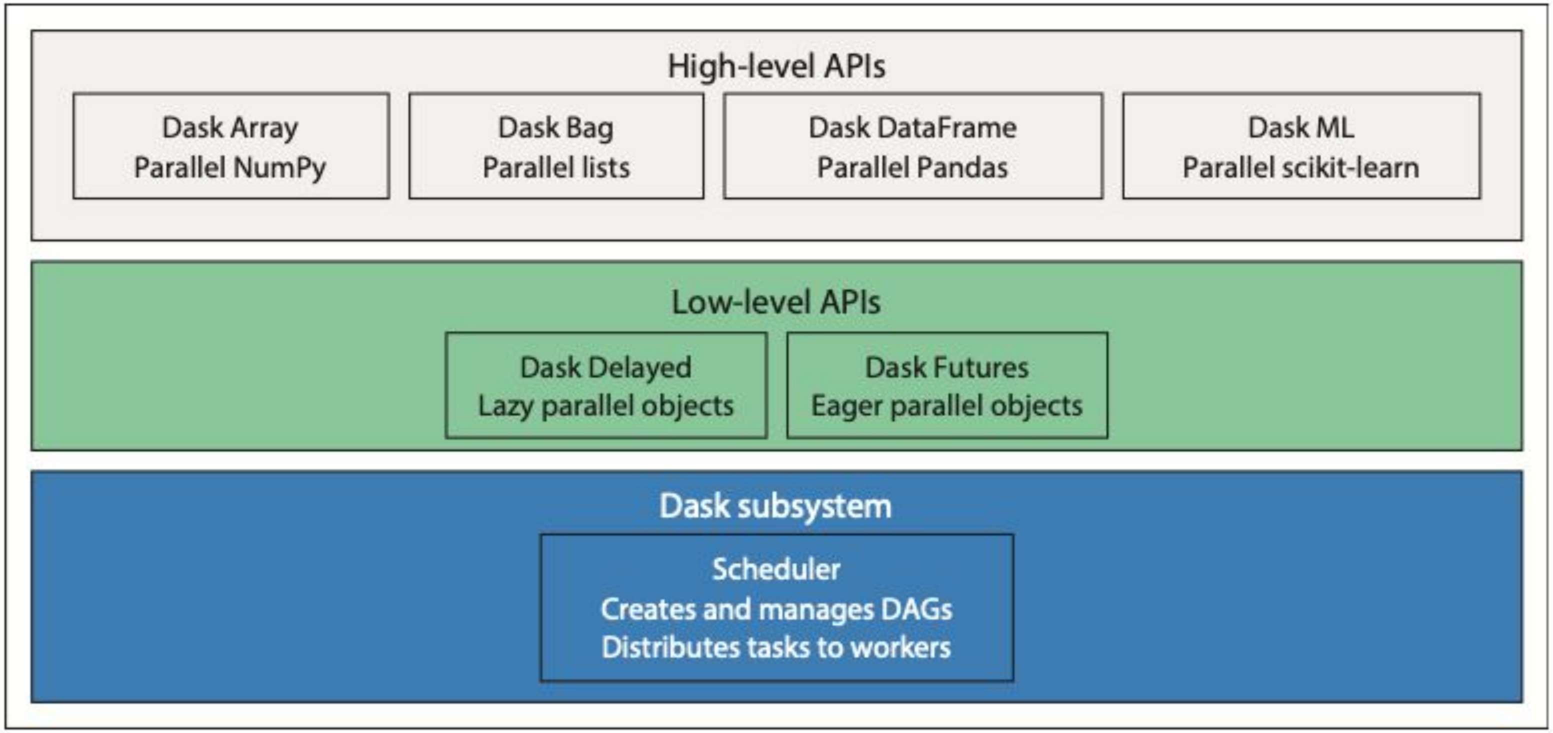
Dataset Size Issues

- **Small datasets**
  - < 1 GB
- **Medium dataset**
  - < 1TB
- **Large dataset**
  - > 1TB
- **The thresholds are fuzzy and changing over time**
  - E.g., you can scale the computer 10x and get 10x bigger data sets
- **Problem when scaling datasets**
  - Long run times
  - Rewriting code in different language / API for datasets of different size
  - Need to think about *what to do* it and *how to do* it efficiently
  - Cumbersome framework

Dask

- Dask is written in Python
  - It scales natively Numpy, Pandas, sklearn
  - Dask objects are wrappers (don’t just mirror the interface) objects from the respective libraries
    - Dask DataFrame = composed of several Pandas DataFrame
    - Dask Array = composed of several Pandas numpy array
  - Parallel parts are called "chunks" or "partitions"
    - Are queued to be worked on
    - Shipped between machines
    - Worked locally on a machine
- Pros
  - Users don't need to learn a new language, but can use familiar interfaces
  - Can focus on writing code that is optimized for parallelism
  - Dask does the heavy lifting
- Scaling Dask is easy
  - Users can write a prototype task on a local machines and use a cluster when needed
  - No need to refactor existing code
  - No need to handle cluster-specific issues
    - E.g., resource management, data recovery, data movement
  - Dask runs on multi-core and
  - Dask can use cluster managers
    - E.g., Yarn, Mesos, Kubernetes, AWS ECS

Dask Layers



High-level APIs
Dask Array
Parallel NumPy
Dask Bag
Parallel lists
Dask DataFrame
Parallel Pandas
Dask ML
Parallel scikit-learn
Low-level APIs
Dask Delayed
Lazy parallel objects
Dask Futures
Eager parallel objects
Dask subsystem
Scheduler
Creates and manages DAGs
Distributes tasks to workers

Scaling Up vs Scaling Out

- **Scaling up**
  - = replace equipment with larger, faster equipments
    - E.g., buy a larger pot
    - Replace knife with food processor
  - **Pros**
    - You got better hardware, nothing else needs to change (e.g., code)
  - **Cons**
    - There will be a time where you exceed the capacity of the current machines
    - Cost: more powerful machines are expensive
- **Scaling out**
  - = divides the work between many workers in parallel
    - E.g., hire more cooks
    - Buy more knives
  - **Pros**
    - Task scheduler organizes computation, assigning workers to each task
    - More cost-effective solution since no specialized hardware is needed
  - **Cons**
    - Need to write code to expose parallelism
    - Costs of maintaining a cluster

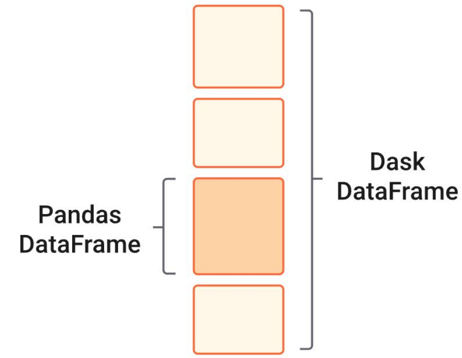
Dask: Computation

- **Lazy computations**
  - User defines the transformations on the data
  - No need to wait for one computation to finish before defining the next
  - Avoid loading the entire data in memory by operating in chunks
  - E.g.,
    - Split a 2GB file into 32 64MB chunks
    - Operate on 8 chunks at a time
    - The max memory consumption doesn't exceed 512MB = (8 x 32)
  - Each task tracks object dimensions and data types
    - No code is executed
- `compute()`
  - Running a computation (aka materializing)

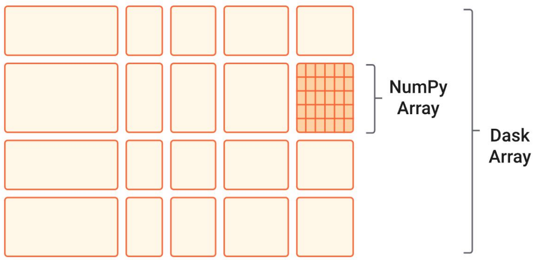
  ```
  missing_count_pct = missing_count.compute()
  ```
- `persist()`
  - As soon as a node in the graph emits results, its intermediate work is
  - discarded to minimize memory usage
  - If we need to do additional computation on intermediate nodes we need to re-run the graph
  - `persist()` tells Dask to keep the intermediate result in memory
  - This speeds up a large and complex DAG that needs to be reused many times

Dask: Data Structures

- **Dask DataFrame** implements Pandas DataFrame
  - Tabular / relational data

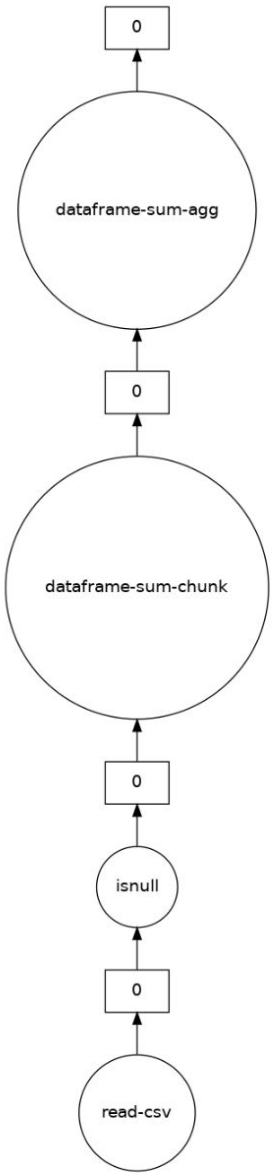


- **Dask Array** implements numpy ndarray
  - Multidimensional array



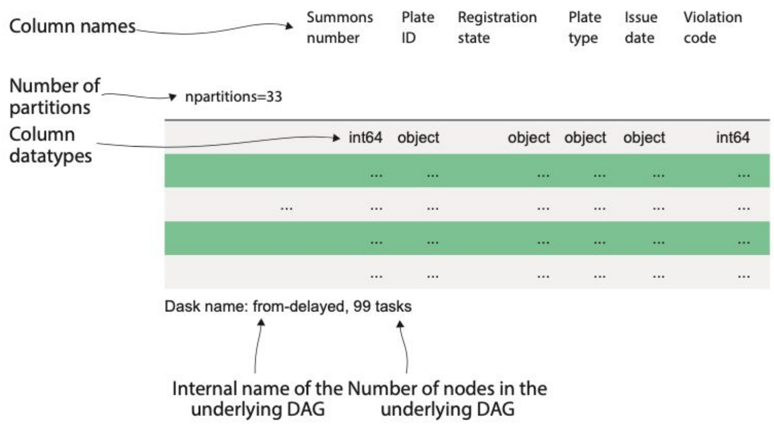
- **Dag Bag** coordinates Python lists of objects
  - Parallelize computations on unstructured or semi-structured data

Dask Reading Data




```python
import dask.dataframe as dd
df = dd.read_csv('nyc-parking-tickets-2017.csv')
missing_values = df.isnull().sum()
missing_values
```

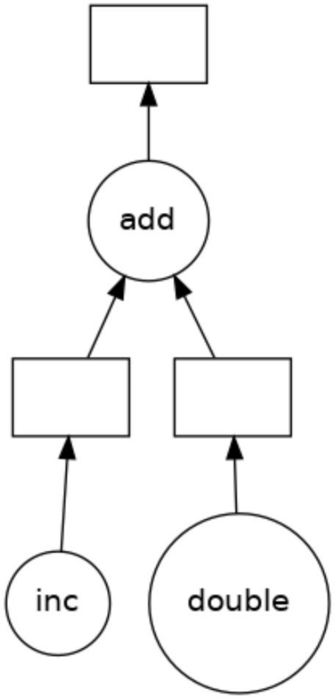
- `dask.dataframe.read_csv()`
  - Doesn't load the data in memory with
  - Tries to infer the types of the columns
    - By randomly sampling some data
    - Best to explicitly set the data types
    - Even better is to use Parquet since it stores data and types together
- Partitions = chunks of data that can be worked independently
  - E.g., 33 partitions
  - Graph is composed of 99 tasks
  - Each partition reads data, splits data, initializes df object



Low Level APIs: Delayed

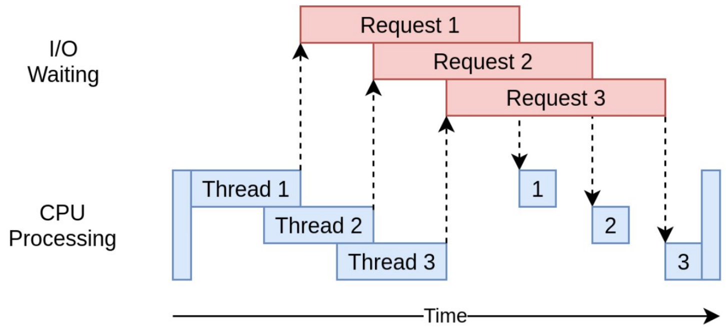
- Handle computations that don’t fit in native Dask data structures (e.g., Dask DataFrame)
- In the example below there is parallelism that can be exploited

```python
def inc(x):
    return x + 1

def double(x):
    return x * 2

def add(x, y):
    return x + y

data = [1, 2, 3, 4, 5]

output = []
for x in data:
    # (x + 1) + (x * 2) = 3x + 1
    a = inc(x)
    b = double(x)
    c = add(a, b)
    # 1 -> 4
    # 2 -> 7
    # 3 -> 10
    # 4 -> 13
    # 5 -> 16
    output.append(c)

# 4 + 7 + 10 + 13 + 16 = 20 + 20 + 10 = 50
total = sum(output)
print(total)
```



Low Level APIs: Futures

- In parallel programming, a "future" encapsulates the asynchronous execution of a callable, representing the eventual result of the operation
- Futures is the most general way of specifying concurrency in Dask
  - Everything can be expressed in terms of futures
  - User can specify what's blocking and what's not blocking
- Python `concurrent.futures`
  - High-level interface for asynchronously executing callables
  - Thread pool or Process pool (same interface `Executor`)
- Dask extends `concurrent.futures`
  - Dask client can be used anywhere `concurrent.futures` can be used




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

def add(x, y):
    return x + y

a = client.submit(inc, 10)
b = client.submit(inc, 20)
```

```
>>> a
<Future: status: pending, key: inc-b8aaf26b99466a7a1980efa1ade6701d>
```

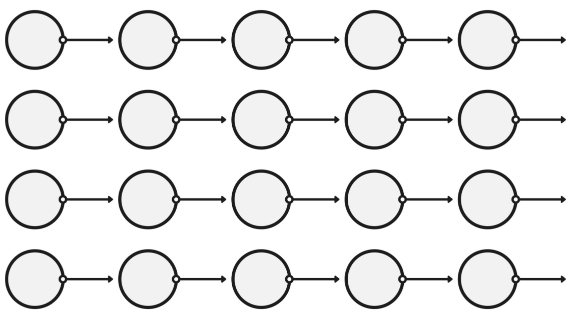
```
>>> a
<Future: status: finished, type: int, key: inc-b8aaf26b99466a7a1980efa1ade6701d>

>>> a.result()  # blocks until task completes and data arrives
11
```

Different Types of Parallel Workload

- Break program in medium-size tasks of computation
  - E.g., a function call

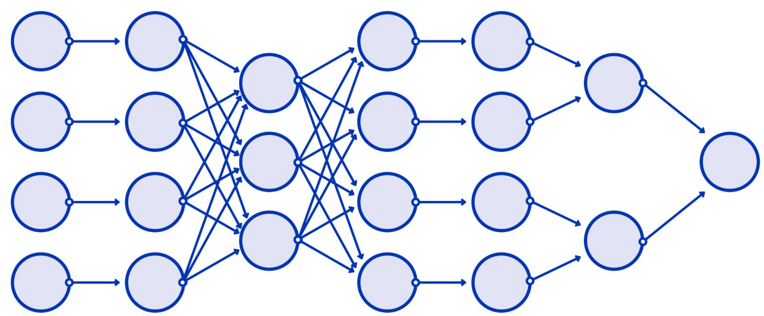
Embarrassingly Parallel

Hadoop/Spark/Dask/Airflow/Prefect



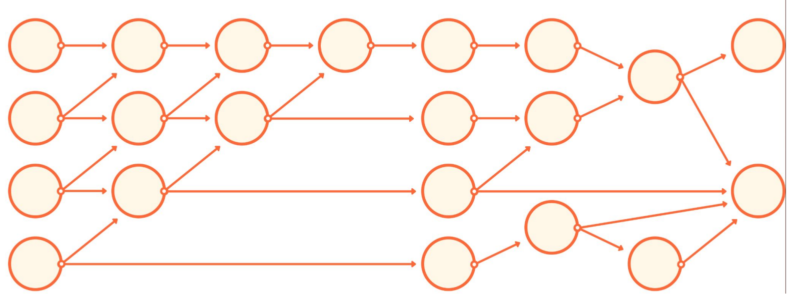
MapReduce

Hadoop/Spark/Dask



Full Task Scheduling

Dask/Airflow/Prefect



Encoding Task Graph

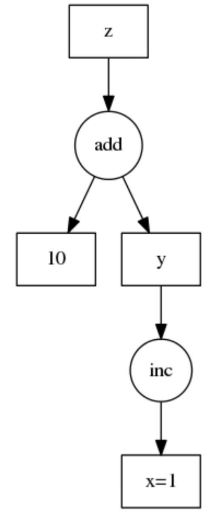
- Dask encodes tasks in terms of Python dicts and functions

```python
def inc(i):
    return i + 1

def add(a, b):
    return a + b

x = 1
y = inc(x)
z = add(y, 10)
```

→

```python
d = {'x': 1,
     'y': (inc, 'x'),
     'z': (add, 'y', 10)}
```

→



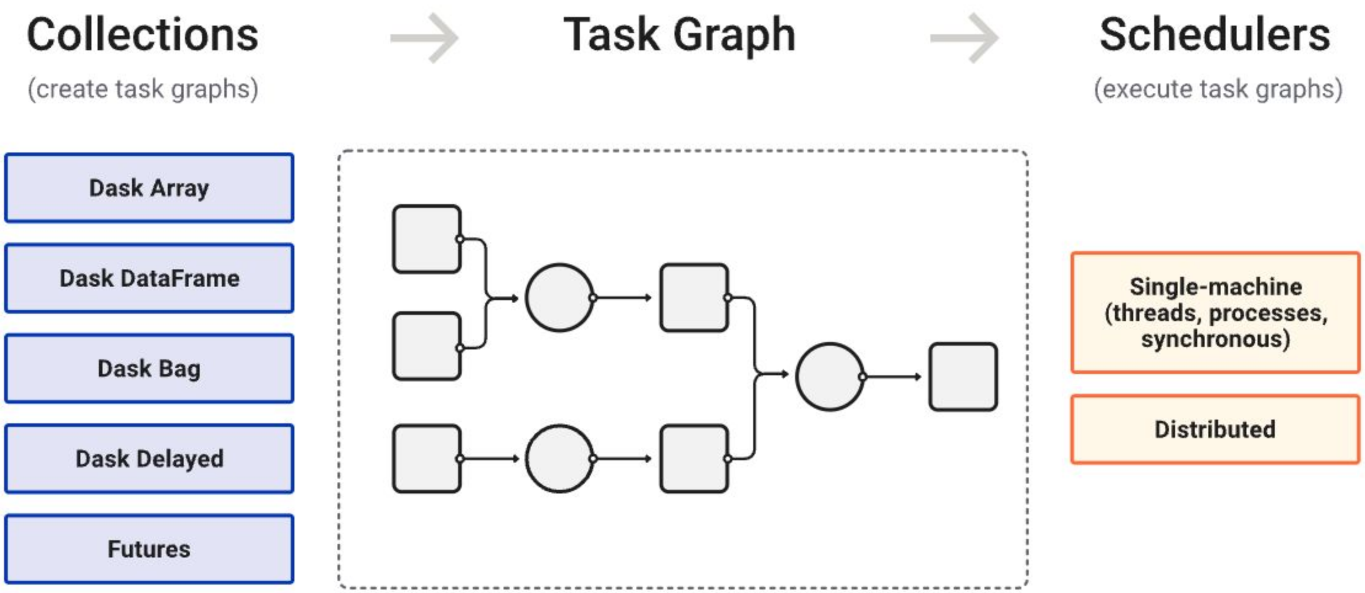

```python
import dask.dataframe as dd

df = dd.read_csv('myfile.*.csv')
df = df + 100
df = df[df.name == 'Alice']
```

→

```python
{
 # From the dask.dataframe.read_csv call
 ('read-csv', 0): (pandas.read_csv, 'myfile.0.csv'),
 ('read-csv', 1): (pandas.read_csv, 'myfile.1.csv'),
 ('read-csv', 2): (pandas.read_csv, 'myfile.2.csv'),
 ('read-csv', 3): (pandas.read_csv, 'myfile.3.csv'),

 # From the df + 100 call
 ('add', 0): (operator.add, ('read-csv', 0), 100),
 ('add', 1): (operator.add, ('read-csv', 1), 100),
 ('add', 2): (operator.add, ('read-csv', 2), 100),
 ('add', 3): (operator.add, ('read-csv', 3), 100),

 # From the df[df.name == 'Alice'] call
 ('filter', 0): (lambda part: part[part.name == 'Alice'], ('add', 0)),
 ('filter', 1): (lambda part: part[part.name == 'Alice'], ('add', 1)),
 ('filter', 2): (lambda part: part[part.name == 'Alice'], ('add', 2)),
 ('filter', 3): (lambda part: part[part.name == 'Alice'], ('add', 3)),
}
```

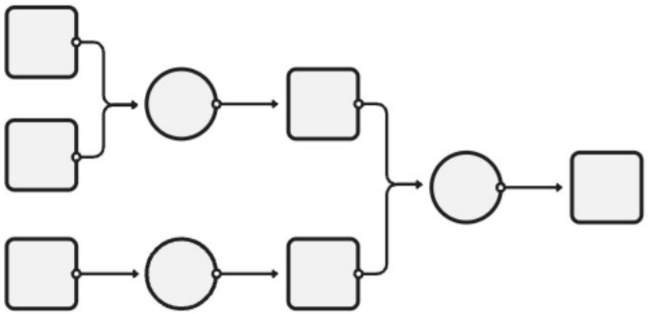
Task Scheduling

- Data collections (Bags, Arrays, DataFrame) and their operations create task graphs
  - Nodes in the task graph are Python functions
  - Edges are dependencies (e.g., output from one task used as input in another task)
- Task graphs are scheduled for execution
- Single-machine scheduler
  - Use local process or thread pool
  - Simple but it can only run on a single machine
- Distributed scheduler
  - It can run locally or distributed across a cluster



Task Scheduling

- **Dask task scheduler orchestrates the work dynamically**
  - Not a static scheduling of operations like a relational DB
  - When the computation takes place, Dask dynamically assesses:
    - What tasks has been completed
    - What tasks is left to do
    - What resources (CPUs) are free
    - Where the data is located
- **This dynamic approach handles a variety issues:**
  - Worker failure
    - Just re-run
  - Workers completing work at different speeds because of:
    - Different computation
    - Different hardware
    - Different workloads on the servers
    - Slower access to the data
  - Network unreliability
    - Just re-run or remove the isolated nodes



Dask vs Spark

- Spark has
- **Pros**
  - Popular framework for analyzing large datasets
  - In-memory alternative to MapReduce / Hadoop
- **Cons**
  - Spark is a Java library, supporting Python through PySpark API
    - Python code is executed on JVM through `py4j`
    - Difficult to debug since execution occurs outside Python
  - Different DataFrame API than Pandas
    - Learn how to do things "the Spark way"
    - You might need to implement things twice to go from exploratory analysis to large experiments / production
  - Optimized for MapReduce operations over a collection
  - Difficult to set-up and configure

Tutorial

Tutorial

- From the official documentation https://docs.dask.org/en/stable/10-minutes-to-dask.html