Scientific Software Development with Python

Parallel computing with Dask

Simon Pfreundschuh
Department of Space, Earth and Environment

CHALMERS
UNIVERSITY OF TECHNOLOGY

1. Introduction

2. Example: Processing SMHI radar images

3. Package managers and compute environments

4. Parallel computing with Dask

- No retrospective next time
- Instead I will finish course content
- Instead final presentations in January

In this lecture

1. Example: Processing SMHI radar images
   - Parallelize task using IPython parallel and Dask
2. Package managers and compute environments
   - Differences between Conda and pip
   - Conda basiscs
3. Parallel computing with Dask
   - Basic parallel computing with Dask
   - Computational graphs
   - Dask array computations

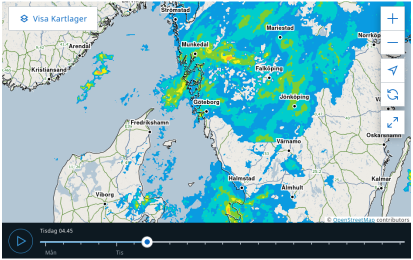
SMHI rain radar data

- SMHI provides access to composite radar images over Sweden
- Download of daily data (288 files, zipped)
- Pixel values x can be converted to radar reflectivity:

$$\text{dBZ} = 0.4x - 30 \qquad (1)$$

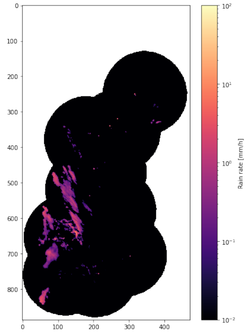
- Radar reflectivity can be converted to rain rate:

$$\text{rr} = \left(\frac{10^{\frac{\text{dBZ}}{10}}}{200}\right)^{\frac{2}{3}} \ [\text{mm h}^{-1}] \qquad (2)$$



Task: Create GIF of monthly precipitation

- Size of single image 886 $\times$ 471.
- 14 GB of data for 1 month.
- Subsample spatial and temporal resolution:
  - average to 8 images per day
  - Subsample images by a factor of 2 along width and height



Predefined code

```
class SMHIRadarImages(collections.abc.Iterable):
    """
    Provides access to all SMHI radar images for a given day.

    Iterating over an SMHIRadarImages object will yield the radar data converted
    to rain rates.

    Attributes:
        file(``zipfile.ZipFile``): The zipfile object containing the images.
        image_files: The list of filenames of the image files in the zipfile.
        n_images(``int``): The number of images of the day.
    """
    def __init__(self, year, month, day):
        """
        Create SMHIRadarImages object for given date.
        """
```

Predefined code

- Convert image data from one day to array:

```
def average_images(smhi_images, n_frames=8):
    """
    Bins images from one day into n_frames temporal bins and calculates
    the averages over each bin.

    Args:
        smhi_images(SMHIImages): SMHIImages object providing access to the
            rain rates for a given day.
        n_frames(int): Into how many frames to bin the data for the given day.
    Returns:
        3-dimensional numpy.ndarray containing the different frames along the
        first axis and the radar composite along the following two.
    """
    ...
```

- Create animation from array:

```
def create_animation(data):
"""
Creates an animation from a 3D array of rain rate data.

Args:
    data(numpy.ndarray): 3D array containing different images along the
         first axis and the image dimensions alsong the second and third.

Returns:
    matplotlib.animation.ArtisAnimation object containing the animated radar
```

- Exercise 1 from notebook
- Time: 10 minutes

The problem

- Manually installing packages is tedious and doesn't scale.
- The more packages you use, the harder it gets to satisfy all their dependencies.

The solution

- Use a program to install other programs (**package manager**)
- Install dependencies separately for each project (**compute environment** or just **environment**)

`pip`

- Official Python package manager
- Supports environments via the `venv` module [1].
- Can only install Python packages[2].

`conda`

- Package and environment manager
- Not limited to Python packages

[1] `venv` the standard environment manager for Python 3.
[2] And thus cannot easily be used to install non-Python dependencies.

Concepts

- `Conda` allows installing packages from different channels (package indices), similar to `pip`
- Packages are distributed in binary format, so no compilation necessary

Installation

- Follow instructions on https://docs.conda.io/projects/conda/en/latest/user-guide/install/index.html

Installing packages

```
conda install numpy
```

Managing environments

- Creating an environment:

```
conda create --name ssdp
```

- Activating an environment:

```
conda activate ssdp
```

- Deactivating an environment:

```
conda deactivate ssdp
```

Determining current environment

```
conda info --envs
```

- Shows defined environments with the currently active one marked with $*$:

```
base                        /home/simon/build/anaconda3
ssdp                     *  /home/simon/build/anaconda3/envs/ssdp
```

Exporting and sharing environments

- Environments can be shared with others by exporting them into a `.yml` file:

```
conda env export > ssdp_conda.yml
```

- To create an environment from a `.yml` file shared with you use

```
conda env create -f ssdp_conda.yml
```

CHALMERS
UNIVERSITY OF TECHNOLOGY

How Conda works

- Conda works through manipulating the system paths which are searched for executable and libraries.
- These settings are handled through environment variables, which are process specific
- Example:

```
$ conda activate base # Activate base environment
$ which python
/home/simon/build/anaconda3/bin/python
$ conda activate ssdp # Activate ssdp environment
$ which python
/home/simon/build/anaconda3/envs/ssdp/bin/python
```

Consequences

- Environments need to be activated for every command line window you open.

- Exercise 2 and 3 from notebook
- Time: 5 + 15 minutes

Dask

- High-level parallel computing library
- Features:
  - Distributed container types (bags, arrays, DataFrames)
  - Builds computational graphs before execution
  - Can run on single host as well as on distributed systems

Advantages

- Similar to IPython parallel Dask provides acts as abstraction layer between computations to perform and the hardware where they are performed.
- This allows scaling you applications from 4 threads on your laptop to 1000s of processes in the cloud.
- Allows processing of data that doesn't fit

Creating a client

- Similar to IPyparallel a client object need to be created to connect to a cluster.
- This simple example will create a client that connects to 4 worker processes that run on your local computer:

```
from dask.distributed import Client
client = Client(n_workers=4)
```

Parallelizing calculations

- The `dask.delayed` function can be used to turn a function into a lazy function.
- Applying the delayed function returns a place-holder object representing the calculation.
- Computing the result, requires calling `compute` method of place-holder object.

Serial version

```
from time import sleep
def add(a, b):
    sleep(1)
    return a + b

% time add(add(1, 2), add(3, 4) # Wall time: 3s
```

Parallelizing calculations

- The `dask.delayed` function can be used to turn a function into a lazy function.
- Applying the delayed function returns a place-holder object representing the calculation.
- Computing the result, requires calling `compute` method of place-holder object.

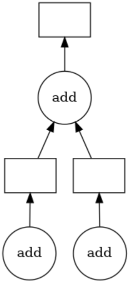
Delayed version

```
add_ = delayed(add)
% time add_(add_(1, 2), add(3, 4)) # Wall time:
% time add_(add_(1, 2), add_(3, 4) # Wall time: 2s
```

Visualizing the computational graph

- In a notebook, the computational graph can be visualized using the `visualize` method:

```
result = add_(add_(1, 2), add_(3, 4))
result.visualize()
```



Parallelizing calculations

- Dask will automatically parallelize independent branches of the computational graph
- This leads to the 1 second speed-up observed in the example.

Building the computational graph

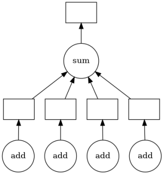
- You can use arbitrary python code to build the computational graph:

```
doubles = []
for i in range(4):
    doubles.append(add_(i, i))

# or
doubles = [add_(i, i) for i in range(4)]

result = delayed(sum)(doubles)
```

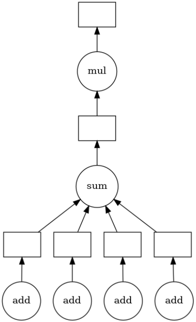
Computational graph



Using `Delayed` objects

- Result of `delayed` functions are represented using `Delayed` objects.
- Accessing calling member functions or accessing attributes of these objects are automatically delayed:

```
double_sum = delayed(sum)(doubles)
# Call of __mul__ member function automatically delayed.
result = double_sum * double_sum
```



Combining computations

```
double_sum = delayed(sum)(double)
result_1 = double_sum * double_sum
result_2 = double_sum + double_sum
```

- Use the `dask.compute` function, when multiple expressions depend on the same parts of a calculations:

```
result_1.compute() # Wall time: 1s
result_2.compute() # Wall time: 1s

a, b = compute(result_1, result_2) # Wall time: 1s
```

Dask bags

- Lazy distributed container
- Functions applied to elements are executed first when `compute` method is called.

Bag example

```
import numpy as np
from dask.bag import from_sequence

bag = from_sequence([10_000] * 100)
random_numbers = bag.map(lambda x: [np.random.rand() for _ in range(x)])
sums = random_numbers.map(sum)
```

Inspecting bag elements

- The `take` method can be used to inspect elements in the bag.
- The elements in the bag are calculated first when requested by the user.

```
print(sums.take(1)) # Prints: (4972.594906446981,)
print(sums.take(1)) # Prints: (4999.976401393778,)
```

Evaluating the list

```
print(sums.compute()) # Prints: [4963.630956357142,
                      #           ...,
                      #           5088.819425189678]
```

- Exercise 3 from notebook
- Time 15 minutes

The real power of Dask: arrays

- Dask arrays let automatically parallelize calculations on large arrays by *blocking*[3].
- This allows you to process data that otherwise wouldn't fit your main memory.

```
import numpy as np
import dask.array as da
x = da.random.rand(size=(10000, 10000),
                   chunks=(1000, 1000))
y = x.mean(axis=0)
```

[3] i.e. split data up into chunks and process separately.

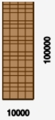
- Calculations are delayed until the `compute` method is called:

```
y.compute()
```

  - In a Jupyter notebook Dask will even visualize the arrays for you:

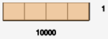
x

	Array	Chunk
Bytes	8.00 GB	80.00 MB
Shape	(100000, 10000)	(4000, 2500)
Count	100 Tasks	100 Chunks
Type	float64	numpy.ndarray



y

	Array	Chunk
Bytes	80.00 kB	20.00 kB
Shape	(10000,)	(2500,)
Count	240 Tasks	4 Chunks
Type	float64	numpy.ndarray



Lazily loading data into array

- Let's assume that we can load 288 images from each day:

```
class SMHIRadarImages(collections.abc.Iterable):
    """
    def __init__(self, year, month, day):
        ...

    def __getitem__(self, i):
        i = min(len(self.image_files) - 1, i)
        filename = self.image_files[i]
        data = io.BytesIO(self.file.read(filename))
        image = PIL.Image.open(data)
        return tif_to_rain_rate(image)
    """
```

- The `__getitem__` function allow us to images via indexing the array:

```
images = SMHIRadarImages(2020, 11, 1)
rain_rates = images[0]
```

Lazily loading data into array

- Let's use Dask to create a list of lazily loaded images:

```
images = map(lambda x: delayed(SMHIRadarImages)(*x), days)
```

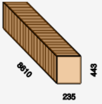
- We can then turn this list into a dask array using `from_delayed` and `da.stack`:

```
for image in images:
  for i in range(288):
    slices.append(da.from_delayed(image[i], shape=(443, 235), dtype=np.float32))
data_array = da.stack(slices, axis=0)
```

Lazily loading data into array

- This gives us the following data array[4]:

	Array	Chunk
Bytes	3.59 GB	416.42 kB
Shape	(8610, 443, 235)	(1, 443, 235)
Count	25860 Tasks	8610 Chunks
Type	float32	numpy.ndarray



- We can then compute statistics of the array[5]:

```
statistics = [da.nanmin(data_array, axis=0),
              da.nanmax(data_array, axis=0),
              da.nanmean(data_array, axis=0),
              da.nanstd(data_array, axis=0)]
```

- And the combine all computations into one:

```
rr_min, rr_max, rr_mean, rr_std = compute(*statistics)
```

[4] Note that we have not downloaded any data, yet.
[5] Without actually computing them, of course.

Parallel computation with Dask

- Dask provides a more high-level interface for parallel computations than IPythonParallel
- Working with lazy operations may need some time to get used to but is extremely powerful.
- This was only a very brief introduction, there's of course a lot more to learn.