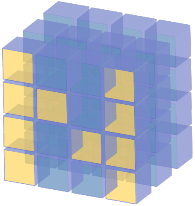
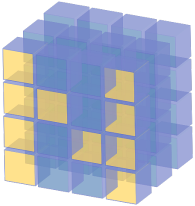
High Performance Computing with Python

Rafael Sarmiento, Vasileios Karakasis, Theofilos Manitaras, and Tim Robinson
ETHZürich / CSCS

Lugano, 11-13.11.2019



NumPy *is a Python library that adds support for large multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.*



- `numpy.ndarray`: a powerful N-dimensional array object
- Sophisticated functions often written in C
- Linear algebra, Fourier transform, and random number capabilities
- Tools for easy binding to Fortran code (F2PY)
- Compatibility with C

The SciPy *library provides many user-friendly and efficient numerical routines for operations such as numerical integration, interpolation, optimization, linear algebra and statistics. SciPy builds on the* `numpy.ndarray` *and expands the set of mathematical functions included in NumPy*

numexpr

numexpr *is a fast numerical expression evaluator for NumPy. It accelerates expressions that operate on arrays like* `3 * a + 4 * b` *and use less memory than doing the same calculation in Python.*

numexpr

```
import numpy as np
import numexpr as ne

x = np.random.random((5000, 50))

ne.evaluate('x**3 + x**2 + x + 1.')
```

numexpr

```python
import numpy as np
import numexpr as ne

x = np.random.random((5000, 50))

ne.evaluate('sin(x) + exp(x)')
```

numexpr

```python
import numpy as np
import numexpr as ne

x = np.random.random((5000, 50))
# x.sum(axis=1)

ne.evaluate('sum(x, axis=1)')[†]
```

[†] According to the [issue#73] ”calculation of sum is slow and uses only one core”, there seems to be a problem with `numexpr`’s reduction with `sum`. Maybe it’s a good idea to avoid reductions with `numexpr`.

numexpr

- multi-threading capabilities
- Sophisticated block algorithms for optimal use of the CPU cache
- Uses MKL for acceleration of transcendental functions

[lab] numexpr

- Let’s open the notebook `numexpr/01_exercise-edm-numpexpr.ipynb`.

We will do the first part together and then you will be asked to modify the `euclidean_trick` function to use numexpr.

numpy.ndarray

```
>>> x = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]], dtype=np.int8)
>>> x
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]], dtype=int8)
```

`numpy.ndarray`

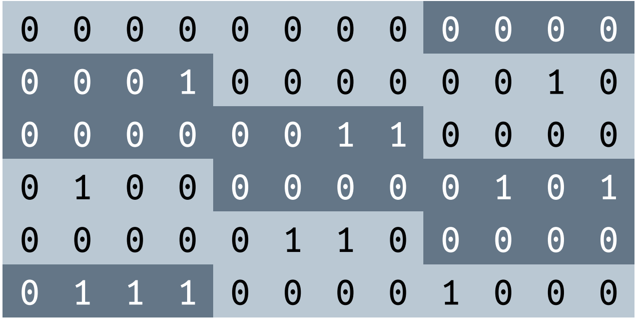
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	1	0
0	0	0	0	0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0	0	1	0	1
0	0	0	0	0	1	1	0	0	0	0	0
0	1	1	1	0	0	0	0	1	0	0	0

`numpy.ndarray`

0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	1	0
0	0	0	0	0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0	0	1	0	1
0	0	0	0	0	1	1	0	0	0	0	0
0	1	1	1	0	0	0	0	1	0	0	0

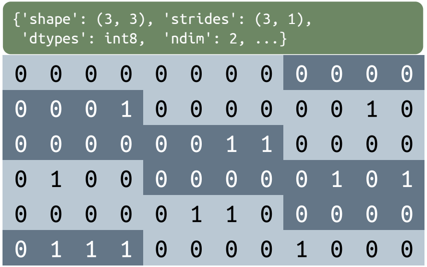
`numpy.ndarray`

```
{'shape': (3, 3), 'strides': (3, 1),
 'dtypes': int8,  'ndim': 2, ...}
```



```
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 1 0
0 0 0 0 0 0 1 1 0 0 0 0
0 1 0 0 0 0 0 0 0 1 0 1
0 0 0 0 0 1 1 0 0 0 0 0
0 1 1 1 0 0 0 0 1 0 0 0
```

`numpy.ndarray`



```
{'shape': (3, 3), 'strides': (3, 1),
 'dtypes': int8,  'ndim': 2, ...}
```

```
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 1 0
0 0 0 0 0 0 1 1 0 0 0 0
0 1 0 0 0 0 0 0 0 1 0 1
0 0 0 0 0 1 1 0 0 0 0 0
0 1 1 1 0 0 0 0 1 0 0 0
```

- The memory block is called **data buffer**.
- The **metadata** is used to interpret the data buffer within the python context.
- The data buffer is stored in C order (row major) by default.
- All items in the array have the same data type.

`numpy.ndarray`

#	#	0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15	#	#

`numpy.ndarray`

Data buffer

#	#	0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15	#	#

NumPy representation

0	1	2	3
4	5	6	7
8	9	10	1
12	12	14	15

```
strides = (4, 1)
shape   = (4, 4)
dtype   = int8
```

numpy.ndarray

Data buffer

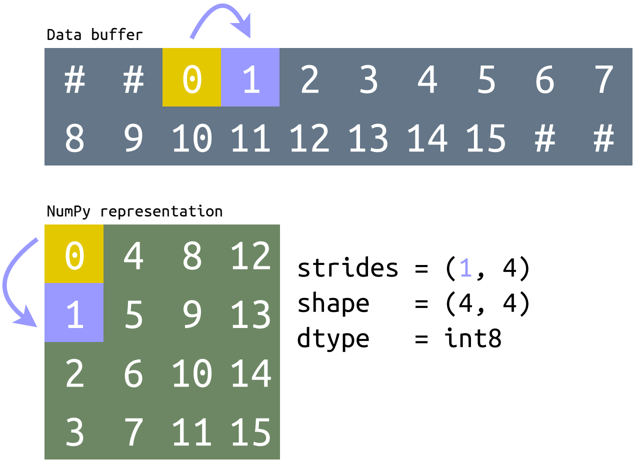
#	#	0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15	#	#

NumPy representation

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

```
strides = (1, 4)
shape   = (4, 4)
dtype   = int8
```

numpy.ndarray



Data buffer

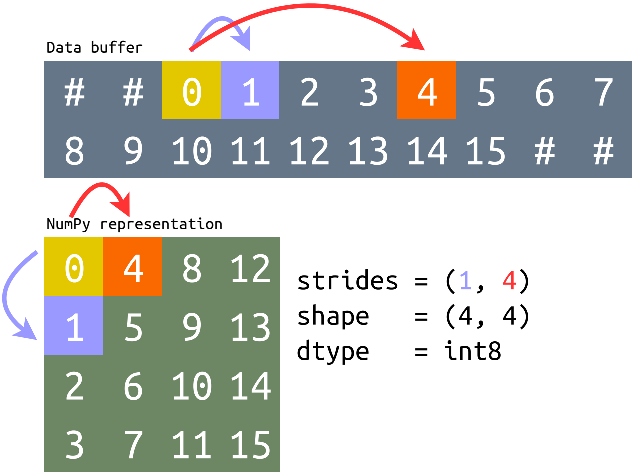
#	#	0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15	#	#

NumPy representation

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

```
strides = (1, 4)
shape   = (4, 4)
dtype   = int8
```

`numpy.ndarray`



Data buffer

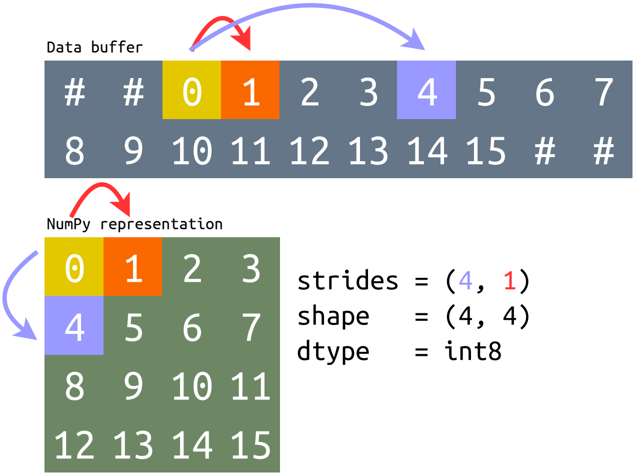
#	#	0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15	#	#

NumPy representation

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

```
strides = (1, 4)
shape   = (4, 4)
dtype   = int8
```

numpy.ndarray



Data buffer

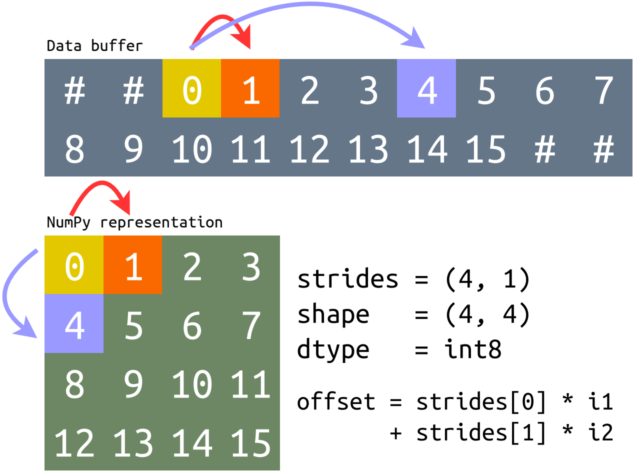
#	#	0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15	#	#

NumPy representation

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

```
strides = (4, 1)
shape   = (4, 4)
dtype   = int8
```

`numpy.ndarray`



Data buffer

#	#	0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15	#	#

NumPy representation

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

```
strides = (4, 1)
shape   = (4, 4)
dtype   = int8

offset = strides[0] * i1
       + strides[1] * i2
```

[lab] `numpy.ndarray` internals

- Let’s open the notebook `numpy/01-numpy-array-internals.ipynb` and go over the questions.

There are two parts:

- **Understanding strides**: a few arrays are given and you are asked to determine the corresponding strides (without looking at the `strides` attribute)
- **Metadata modification vs copying the data buffer**: some operations are given and we ask you to explain the results or differences in execution time. (Hint: Indentify if new data is created or if the operations can be done with only a change of metadata)

Broadcasting

$$\begin{bmatrix} x_0 & x_1 & x_2 & \cdots & x_n \end{bmatrix} + \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

$$(1, n) \qquad\qquad (n, 1)$$

Broadcasting

$$\underset{(1,\, n)}{\begin{bmatrix} x_0 & x_1 & x_2 & \cdots & x_n \end{bmatrix}} + \underset{(n,\, 1)}{\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}} = \underset{(n,\, n)}{\begin{bmatrix} y_0 + x_0 & y_0 + x_1 & \cdots & y_0 + x_n \\ y_1 + x_0 & y_1 + x_1 & \cdots & y_1 + x_n \\ y_2 + x_0 & y_2 + x_1 & \cdots & y_2 + x_n \\ \vdots & \vdots & \ldots & \vdots \\ y_n + x_0 & y_n + x_1 & \cdots & y_n + x_n \end{bmatrix}}$$

[lab] Broadcasting

- Let’s open the notebook `numpy/02-broadcasting.ipynb` and go over the cells and the questions. The goal of this notebook is to understand the two broadcasting operations presented there.

Vectorization

- Use operations over the whole array instead of over single elements.

Vectorization

- Use operations over the whole array instead of over single elements.

```
x = np.array([0., 1., 2., 3., 4., 5.])
y = x[x % 2 == 0]  # y = x[array([ True, False,  True,
                   #               False,  True, False])]
# y = array([0., 2., 4.])
sq_mod_even = np.dot(y, y)
```

Vectorization

- Use operations over the whole array instead of over single elements.

```
x = np.array([0., 1., 2., 3., 4., 5.])
y = x[x % 2 == 0]  # y = x[array([ True, False,  True,
                   #                False,  True, False])]
# y = array([0., 2., 4.])
sq_mod_even = np.dot(y, y)
```

- When working with arrays, use *ufuncs* and general NumPy's functions.

```
x = np.exp(y)      # array([1.00e+00, 7.39e+00, 5.46e+01])
p = np.dot(x, y)
```

Vectorization

- Use operations over the whole array instead of over single elements.

```
x = np.array([0., 1., 2., 3., 4., 5.])
y = x[x % 2 == 0]  # y = x[array([ True, False,  True,
                   #             False,  True, False])]
# y = array([0., 2., 4.])
sq_mod_even = np.dot(y, y)
```

- When working with arrays, use *ufuncs* and general NumPy’s functions.

```
x = np.exp(y)      # array([1.00e+00, 7.39e+00, 5.46e+01])
p = np.dot(x, y)
```

- Adapt your solutions to use the two points above.

Euclidean distance matrix

$$d_{\mathsf{e}}\left(\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ \dots & \dots & \dots \\ x_{n1} & x_{n2} & x_{n3} \end{bmatrix}, \begin{bmatrix} y_{11} & y_{12} & y_{13} \\ y_{21} & y_{22} & y_{23} \\ \dots & \dots & \dots \\ y_{n1} & y_{n2} & y_{n3} \end{bmatrix}\right) = \begin{bmatrix} \sum (x_{1i} - y_{1i})^2 & \sum (x_{1i} - y_{2i})^2 & \dots & \sum (x_{1i} - y_{ni})^2 \\ \sum (x_{2i} - y_{1i})^2 & \sum (x_{2i} - y_{2i})^2 & \dots & \sum (x_{2i} - y_{ni})^2 \\ \dots & \dots & \dots & \dots \\ \sum (x_{ni} - y_{1i})^2 & \sum (x_{ni} - y_{2i})^2 & \dots & \sum (x_{ni} - y_{ni})^2 \end{bmatrix}$$

Euclidean distance matrix

$$d_{\mathsf{e}}\left(\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ \dots & \dots & \dots \\ x_{n1} & x_{n2} & x_{n3} \end{bmatrix}, \begin{bmatrix} y_{11} & y_{12} & y_{13} \\ y_{21} & y_{22} & y_{23} \\ \dots & \dots & \dots \\ y_{n1} & y_{n2} & y_{n3} \end{bmatrix}\right) = \begin{bmatrix} \sum(x_{1i} - y_{1i})^2 & \sum(x_{1i} - y_{2i})^2 & \dots & \sum(x_{1i} - y_{ni})^2 \\ \sum(x_{2i} - y_{1i})^2 & \sum(x_{2i} - y_{2i})^2 & \dots & \sum(x_{2i} - y_{ni})^2 \\ \dots & \dots & \dots & \dots \\ \sum(x_{ni} - y_{1i})^2 & \sum(x_{ni} - y_{2i})^2 & \dots & \sum(x_{ni} - y_{ni})^2 \end{bmatrix}$$

Euclidean distance matrix

$$d_{\text{e}}\left(\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ \dots & \dots & \dots \\ x_{n1} & x_{n2} & x_{n3} \end{bmatrix}, \begin{bmatrix} y_{11} & y_{12} & y_{13} \\ y_{21} & y_{22} & y_{23} \\ \dots & \dots & \dots \\ y_{n1} & y_{n2} & y_{n3} \end{bmatrix}\right) = \begin{bmatrix} \sum (x_{1i} - y_{1i})^2 & \sum (x_{1i} - y_{2i})^2 & \dots & \sum (x_{1i} - y_{ni})^2 \\ \sum (x_{2i} - y_{1i})^2 & \sum (x_{2i} - y_{2i})^2 & \dots & \sum (x_{2i} - y_{ni})^2 \\ \dots & \dots & \dots & \dots \\ \sum (x_{ni} - y_{1i})^2 & \sum (x_{ni} - y_{2i})^2 & \dots & \sum (x_{ni} - y_{ni})^2 \end{bmatrix}$$

Euclidean distance matrix

$$d_{\mathsf{e}}\left(\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ \dots & \dots & \dots \\ x_{n1} & x_{n2} & x_{n3} \end{bmatrix}, \begin{bmatrix} y_{11} & y_{12} & y_{13} \\ y_{21} & y_{22} & y_{23} \\ \dots & \dots & \dots \\ y_{n1} & y_{n2} & y_{n3} \end{bmatrix}\right) = \begin{bmatrix} \sum (x_{1i} - y_{1i})^2 & \sum (x_{1i} - y_{2i})^2 & \dots & \sum (x_{1i} - y_{ni})^2 \\ \sum (x_{2i} - y_{1i})^2 & \sum (x_{2i} - y_{2i})^2 & \dots & \sum (x_{2i} - y_{ni})^2 \\ \dots & \dots & \dots & \dots \\ \sum (x_{ni} - y_{1i})^2 & \sum (x_{ni} - y_{2i})^2 & \dots & \sum (x_{ni} - y_{ni})^2 \end{bmatrix}$$

Euclidean distance matrix

$$d_{\mathsf{e}}\left(\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ \dots & \dots & \dots \\ x_{n1} & x_{n2} & x_{n3} \end{bmatrix}, \begin{bmatrix} y_{11} & y_{12} & y_{13} \\ y_{21} & y_{22} & y_{23} \\ \dots & \dots & \dots \\ y_{n1} & y_{n2} & y_{n3} \end{bmatrix}\right) = \begin{bmatrix} \sum(x_{1i}-y_{1i})^2 & \sum(x_{1i}-y_{2i})^2 & \dots & \sum(x_{1i}-y_{ni})^2 \\ \sum(x_{2i}-y_{1i})^2 & \sum(x_{2i}-y_{2i})^2 & \dots & \sum(x_{2i}-y_{ni})^2 \\ \dots & \dots & \dots & \dots \\ \sum(x_{ni}-y_{1i})^2 & \sum(x_{ni}-y_{2i})^2 & \dots & \sum(x_{ni}-y_{ni})^2 \end{bmatrix}$$

Euclidean distance matrix

$$d_{\text{e}}\left(\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ \dots & \dots & \dots \\ x_{n1} & x_{n2} & x_{n3} \end{bmatrix}, \begin{bmatrix} y_{11} & y_{12} & y_{13} \\ y_{21} & y_{22} & y_{23} \\ \dots & \dots & \dots \\ y_{n1} & y_{n2} & y_{n3} \end{bmatrix}\right) = \begin{bmatrix} \sum\left(x_{1i} - y_{1i}\right)^2 & \sum\left(x_{1i} - y_{2i}\right)^2 & \dots & \sum\left(x_{1i} - y_{ni}\right)^2 \\ \sum\left(x_{2i} - y_{1i}\right)^2 & \sum\left(x_{2i} - y_{2i}\right)^2 & \dots & \sum\left(x_{2i} - y_{ni}\right)^2 \\ \dots & \dots & \dots & \dots \\ \sum\left(x_{ni} - y_{1i}\right)^2 & \sum\left(x_{ni} - y_{2i}\right)^2 & \dots & \sum\left(x_{ni} - y_{ni}\right)^2 \end{bmatrix}$$

Euclidean distance matrix

$$d_{\mathrm{e}}\left(\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ \dots & \dots & \dots \\ x_{n1} & x_{n2} & x_{n3} \end{bmatrix}, \begin{bmatrix} y_{11} & y_{12} & y_{13} \\ y_{21} & y_{22} & y_{23} \\ \dots & \dots & \dots \\ y_{n1} & y_{n2} & y_{n3} \end{bmatrix}\right) = \begin{bmatrix} \sum (x_{1i} - y_{1i})^2 & \sum (x_{1i} - y_{2i})^2 & \dots & \sum (x_{1i} - y_{ni})^2 \\ \sum (x_{2i} - y_{1i})^2 & \sum (x_{2i} - y_{2i})^2 & \dots & \sum (x_{2i} - y_{ni})^2 \\ \dots & \dots & \dots & \dots \\ \sum (x_{ni} - y_{1i})^2 & \sum (x_{ni} - y_{2i})^2 & \dots & \sum (x_{ni} - y_{ni})^2 \end{bmatrix}$$

Euclidean distance matrix

$$d_{\mathsf{e}}\left(\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ \dots & \dots & \dots \\ x_{n1} & x_{n2} & x_{n3} \end{bmatrix}, \begin{bmatrix} y_{11} & y_{12} & y_{13} \\ y_{21} & y_{22} & y_{23} \\ \dots & \dots & \dots \\ y_{n1} & y_{n2} & y_{n3} \end{bmatrix}\right) = \begin{bmatrix} \sum (x_{1i} - y_{1i})^2 & \sum (x_{1i} - y_{2i})^2 & \dots & \sum (x_{1i} - y_{ni})^2 \\ \sum (x_{2i} - y_{1i})^2 & \sum (x_{2i} - y_{2i})^2 & \dots & \sum (x_{2i} - y_{ni})^2 \\ \dots & \dots & \dots & \dots \\ \sum (x_{ni} - y_{1i})^2 & \sum (x_{ni} - y_{2i})^2 & \dots & \sum (x_{ni} - y_{ni})^2 \end{bmatrix}$$

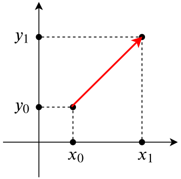
Euclidean distance matrix

$$d_{\mathsf{e}}\left(\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ \dots & \dots & \dots \\ x_{n1} & x_{n2} & x_{n3} \end{bmatrix}, \begin{bmatrix} y_{11} & y_{12} & y_{13} \\ y_{21} & y_{22} & y_{23} \\ \dots & \dots & \dots \\ y_{n1} & y_{n2} & y_{n3} \end{bmatrix}\right) = \begin{bmatrix} \sum (x_{1i} - y_{1i})^2 & \sum (x_{1i} - y_{2i})^2 & \dots & \sum (x_{1i} - y_{ni})^2 \\ \sum (x_{2i} - y_{1i})^2 & \sum (x_{2i} - y_{2i})^2 & \dots & \sum (x_{2i} - y_{ni})^2 \\ \dots & \dots & \dots & \dots \\ \sum (x_{ni} - y_{1i})^2 & \sum (x_{ni} - y_{2i})^2 & \dots & \sum (x_{ni} - y_{ni})^2 \end{bmatrix}$$

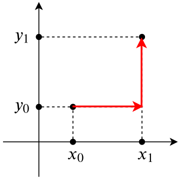
Euclidean distance matrix

$$d_{\mathsf{e}}\left(\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ \dots & \dots & \dots \\ x_{n1} & x_{n2} & x_{n3} \end{bmatrix}, \begin{bmatrix} y_{11} & y_{12} & y_{13} \\ y_{21} & y_{22} & y_{23} \\ \dots & \dots & \dots \\ y_{n1} & y_{n2} & y_{n3} \end{bmatrix}\right) = \begin{bmatrix} \sum(x_{1i} - y_{1i})^2 & \sum(x_{1i} - y_{2i})^2 & \dots & \sum(x_{1i} - y_{ni})^2 \\ \sum(x_{2i} - y_{1i})^2 & \sum(x_{2i} - y_{2i})^2 & \dots & \sum(x_{2i} - y_{ni})^2 \\ \dots & \dots & \dots & \dots \\ \sum(x_{ni} - y_{1i})^2 & \sum(x_{ni} - y_{2i})^2 & \dots & \sum(x_{ni} - y_{ni})^2 \end{bmatrix}$$

Euclidean and Cityblock distance matrices



$$d_{\mathsf{e}}\left(\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ \dots & \dots & \dots \\ x_{n1} & x_{n2} & x_{n3} \end{bmatrix}, \begin{bmatrix} y_{11} & y_{12} & y_{13} \\ y_{21} & y_{22} & y_{23} \\ \dots & \dots & \dots \\ y_{n1} & y_{n2} & y_{n3} \end{bmatrix}\right) = \begin{bmatrix} \sum(x_{1i} - y_{1i})^2 & \sum(x_{1i} - y_{2i})^2 & \dots & \sum(x_{1i} - y_{ni})^2 \\ \sum(x_{2i} - y_{1i})^2 & \sum(x_{2i} - y_{2i})^2 & \dots & \sum(x_{2i} - y_{ni})^2 \\ \dots & \dots & \dots & \dots \\ \sum(x_{ni} - y_{1i})^2 & \sum(x_{ni} - y_{2i})^2 & \dots & \sum(x_{ni} - y_{ni})^2 \end{bmatrix}$$



$$d_{\mathsf{cb}}\left(\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ \dots & \dots & \dots \\ x_{n1} & x_{n2} & x_{n3} \end{bmatrix}, \begin{bmatrix} y_{11} & y_{12} & y_{13} \\ y_{21} & y_{22} & y_{23} \\ \dots & \dots & \dots \\ y_{n1} & y_{n2} & y_{n3} \end{bmatrix}\right) = \begin{bmatrix} \sum|x_{1i} - y_{1i}| & \sum|x_{1i} - y_{2i}| & \dots & \sum|x_{1i} - y_{ni}| \\ \sum|x_{2i} - y_{1i}| & \sum|x_{2i} - y_{2i}| & \dots & \sum|x_{2i} - y_{ni}| \\ \dots & \dots & \dots & \dots \\ \sum|x_{ni} - y_{1i}| & \sum|x_{ni} - y_{2i}| & \dots & \sum|x_{ni} - y_{ni}| \end{bmatrix}$$

```python
def euclidean_distance_matrix(x, y):
    num_samples = x.shape[0]
    dist_matrix = np.empty((num_samples,
                            num_samples))
    for i, xi in enumerate(x):
        for j, yj in enumerate(y):
            diff = xi - yj
            dist_matrix[i][j] = np.dot(diff, diff)

    return dist_matrix
```

```python
def euclidean_distance_matrix(x, y):
    num_samples = x.shape[0]
    dist_matrix = np.empty((num_samples,
                            num_samples))
    for i, xi in enumerate(x):
        for j, yj in enumerate(y):
            diff = xi - yj
            dist_matrix[i][j] = np.dot(diff, diff)

    return dist_matrix
```

- Use operations over the whole array instead of over single elements.

✓ When working with arrays, use ufuncs and general NumPy’s functions.

- Adapt your solutions to use the two points above.

$$\sum_k (x_{ik} - y_{jk})^2 = (\vec{x}_i - \vec{y}_j) \cdot (\vec{x}_i - \vec{y}_j) = \vec{x}_i \cdot \vec{x}_i + \vec{y}_j \cdot \vec{y}_j - 2\vec{x}_i \cdot \vec{y}_j$$

$$\sum_k (x_{ik} - y_{jk})^2 = (\vec{x}_i - \vec{y}_j) \cdot (\vec{x}_i - \vec{y}_j) = \vec{x}_i \cdot \vec{x}_i + \vec{y}_j \cdot \vec{y}_j - 2\vec{x}_i \cdot \vec{y}_j$$

$\vec{x}_i \cdot \vec{y}_j \rightarrow$ `np.dot(x, y)` : Matrix product of $\{\vec{x}\}$ and $\{\vec{y}\}$

$$\sum_k (x_{ik} - y_{jk})^2 = (\vec{x}_i - \vec{y}_j) \cdot (\vec{x}_i - \vec{y}_j) = \vec{x}_i \cdot \vec{x}_i + \vec{y}_j \cdot \vec{y}_j - 2\vec{x}_i \cdot \vec{y}_j$$

$\vec{x}_i \cdot \vec{y}_j \rightarrow$ `np.dot(x, y)` : Matrix product of $\{\vec{x}\}$ and $\{\vec{y}\}$

$\vec{x}_i \cdot \vec{x}_i \rightarrow$ `np.einsum('ij,ij->i', x, x)` : A vector of elements $\sum_j x_{ij}x_{ij} \equiv \sum_j x_{ij}^2$

$\vec{y}_j \cdot \vec{y}_j \rightarrow$ `np.einsum('ij,ij->i', y, y)` : A vector of elements $\sum_j y_{ij}y_{ij} \equiv \sum_j y_{ij}^2$

$$\sum_k (x_{ik} - y_{jk})^2 = (\vec{x}_i - \vec{y}_j) \cdot (\vec{x}_i - \vec{y}_j) = \vec{x}_i \cdot \vec{x}_i + \vec{y}_j \cdot \vec{y}_j - 2\vec{x}_i \cdot \vec{y}_j$$

$\vec{x}_i \cdot \vec{y}_j \rightarrow$ `np.dot(x, y)`

$\vec{x}_i \cdot \vec{x}_i \rightarrow$ `np.einsum('ij,ij->i', x, x)[:, np.newaxis]`

$\vec{y}_j \cdot \vec{y}_j \rightarrow$ `np.einsum('ij,ij->i', y, y)[np.newaxis, :]`

```python
def euclidean_distance_matrix(x, y):
    x2 = np.einsum('ij,ij->i', x, x)[:, np.newaxis]
    y2 = np.einsum('ij,ij->i', y, y)[np.newaxis, :]
    xy = np.dot(x, y.T)

    return np.abs(x2 + y2 - 2. * xy)
```

[lab] Euclidean distance matrix with NumPy

- Let’s open the notebook `euclidean-distance-matrix-numpy.ipynb` and check step by step what the function `euclidean_numpy` does:
  - What’s the shape of the array resulting from the `np.einsum` operation? Why?
  - What’s the effect of adding a new axis to an array with `[:, np.newaxis]`?
  - What’s the effect of adding a new axis to an array with `[np.newaxis, :]`?
  - What’s the effect of the sum `x2 + y2` in the `euclidean_numpy` function?
  - Why is necessary to add a new axis?
- Run all cells and compare the execution times of the different approaches.
- While running the `%timeit` function calls, you may open a terminal and check the load with the command `top`.

Cityblock distance matrix

$$\sum_k |x_{ik} - y_{jk}|$$

The trick we used for the Euclidean distance matrix doesn’t work here!

Numba *is an open source just-in-time (JIT) compiler that translates a subset of Python and NumPy code into fast machine code.*

- Translation of python functions to machine code at runtime using the LLVM compiler library
- Designed to be used with NumPy arrays
- Options to parallelize code for CPUs and GPUs and automatic SIMD Vectorization
- Support for both NVIDIA’s CUDA and AMD’s ROCm driver allowing to write parallel GPU code from Python.

```python
def reduce(x):
    x_sum = 0.0
    for i in range(x.shape[0]):
        x_sum += x[i]

    return x_sum
```

```python
import numba

@numba.jit(nopython=True)
def reduce(x):
    x_sum = 0.0
    for i in range(x.shape[0]):
        x_sum += x[i]

    return x_sum
```

[lab] Cityblock distance matrix with Numba's just-in-time compilation

- Let's run the notebook `numba/simple/cityblock-distance-matrix-numba.jit.ipynb`.
  - Notice that the function to be decorated with `@numba.jit` is not written in a pythonic style. Instead, with the loops it resembles more the C or Fortran styles.
  - What are the diferences between the two Numba implementations?
  - Notice that on the innermost loop we use `numba.prange` to run it in parallel.
- Run all cells and compare the execution times of the different approaches.
- While running the `%timeit` function calls, you may open a terminal and check with `top` that the decorated functions run in multiple threads.

Dask *is a flexible library for parallel computing in Python. It provides dynamic task scheduling optimized for computation as well as big data collections like parallel arrays, dataframes, and lists that extend common interfaces like NumPy and Pandas to larger-than-memory or distributed environments.*

- `dask.array` implements a subset of the NumPy array interface using blocked algorithms, cutting up the large array into chunks of small arrays.
- `dask.bag` parallelizes computations across a large collection of generic Python objects.
- `dask.dataframe` is a large parallel DataFrame composed of many smaller Pandas DataFrames which may live on disk for larger-than-memory computing on a single machine or a cluster.

- `dask.delayed` can be used to parallelize custom algorithms. It allows to create graphs with a light annotation of normal python code:

```
x = dask.delayed(myfunction1)(<args>)
y = dask.delayed(myfunction2)(<args>)
z = dask.delayed(myfunction3)(x, y)
z.compute(scheduler='threads')
```

- `dask.delayed` can be used to parallelize custom algorithms. It allows to create graphs with a light annotation of normal python code:

```
list_delayed = [dask.delayed(myfunction1)(<args>),
                dask.delayed(myfunction2)(<args>),
                dask.delayed(myfunction3)(<args>)]

dask.compute(*list_delayed, scheduler='threads')
```

[lab] Simple Dask graphs

- Let’s run the notebook `dask/01-dask-intro.ipynb`.
  - The goal is the go over the cells and questions and annotate the code with `dask.delayed` to make the execution lazy.
  - Before running predict how much time it will take.
  - The processor on Piz Daint’s ’gpu’ nodes have 24 threads. Try a number of tasks higher and lower than 24 a see what happens.

[lab] Cityblock distance matrix with SciPy and Dask

- Let’s run the notebook `dask/02-exercise-cityblock-distance-matrix-scipy.dask.ipynb`.
- `scipy.spatial.distance.cdist` can be used to compute the cityblock distance matrix. It is fast but doesn’t use OpenMP threads. We can easily write a distributed Cityblock distance matrix function based on `cdist` with the help of Dask.

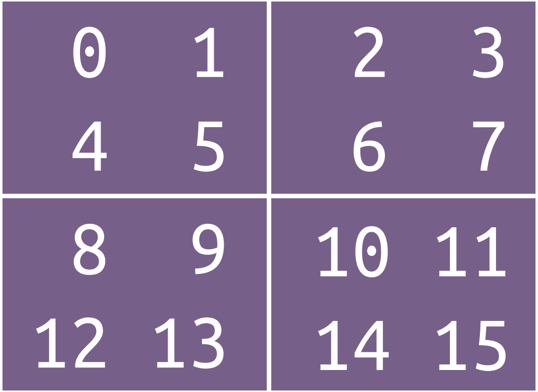
    - Same as the previous exercise, go over the cells and annotate the code to execute it lazily. This time you have to go over the notebook and find what needs to be changed.
    - While timing `cdist` check with `top` that it runs on a single thread.
    - Why is it relevant for the implementation of such distributed function that `cdist` runs on a single thread?
    - Check that when we create the list of delayed functions the execution is deferred to when `compute` is called.

- Run all cells and compare the execution times of the different approaches.
- While running the `%timeit` function calls, you may open a terminal and check with `top` that the new distributed function is runnig in multiple threads.

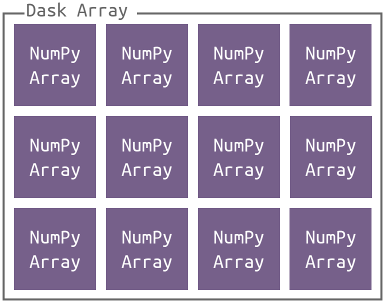
Dask array

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Dask array



0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15



Dask Array

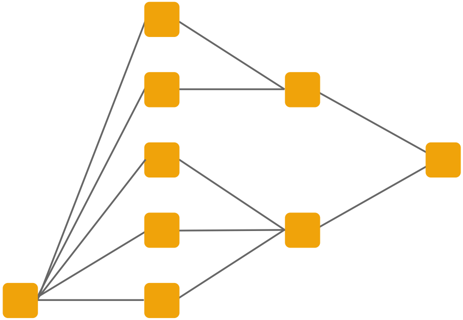
NumPy Array | NumPy Array | NumPy Array | NumPy Array

NumPy Array | NumPy Array | NumPy Array | NumPy Array

NumPy Array | NumPy Array | NumPy Array | NumPy Array

- A Dask array consists of many NumPy arrays arranged into a grid
- Those NumPy arrays may live on memory, disk or remote machines
- `dask.array` implements many of the numpy functions but in block-wise fashion and are executed through a graph.
- For equal sizes, operations on Dask arrays are in general slower than the corresponding NumPy ones.

`dask.array` graph

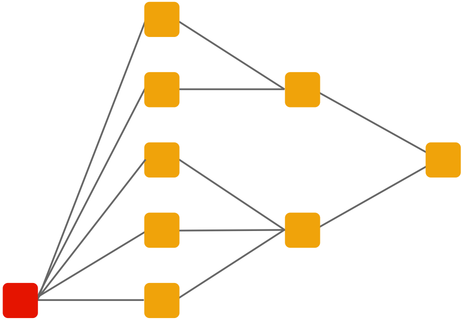


🟧 waiting

🟥 computing/allocated

🟩 memory freed

`dask.array` graph

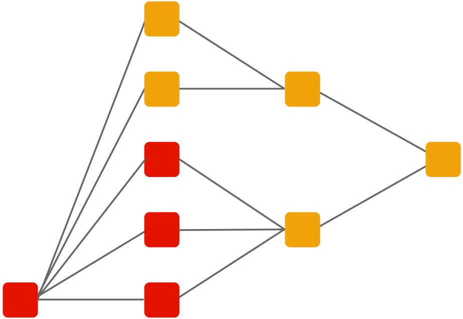


🟧 waiting

🟥 computing/allocated

🟩 memory freed

`dask.array` graph

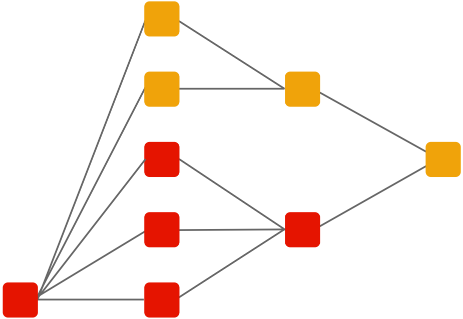


🟧 `waiting`

🟥 `computing/allocated`

🟩 `memory freed`

`dask.array` graph

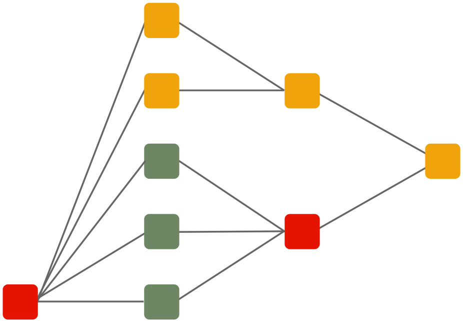


🟧 waiting

🟥 computing/allocated

🟩 memory freed

`dask.array` graph

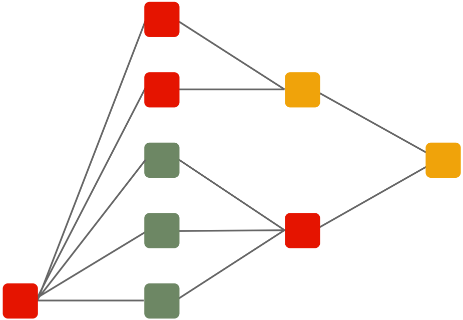


🟧 `waiting`

🟥 `computing/allocated`

🟩 `memory freed`

`dask.array` graph

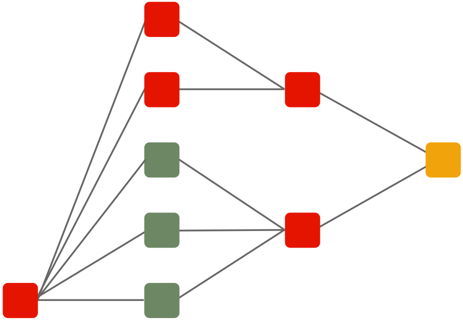


🟧 waiting

🟥 computing/allocated

🟩 memory freed

`dask.array` graph

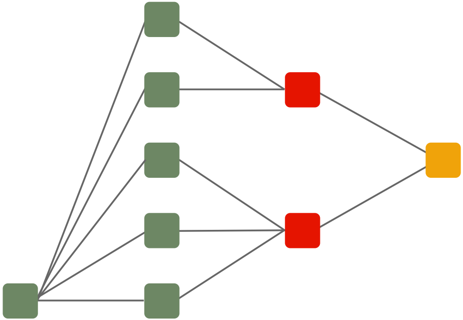


🟧 `waiting`

🟥 `computing/allocated`

🟩 `memory freed`

`dask.array` graph

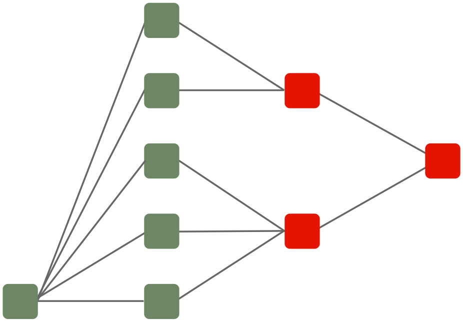


🟧 `waiting`

🟥 `computing/allocated`

🟩 `memory freed`

`dask.array` graph

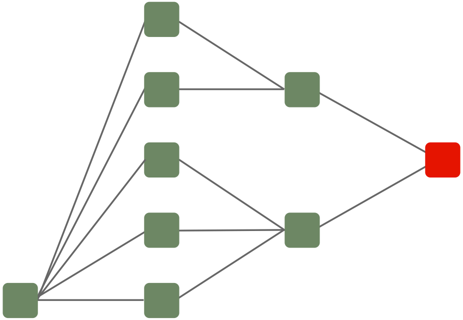


🟧 waiting

🟥 computing/allocated

🟩 memory freed

`dask.array` graph

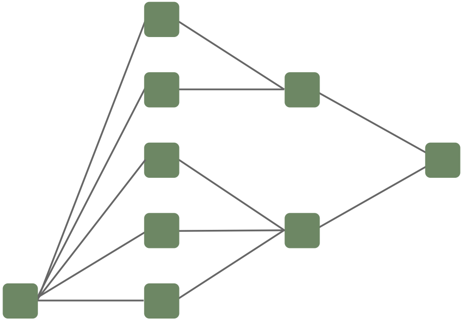


🟧 `waiting`

🟥 `computing/allocated`

🟩 `memory freed`

`dask.array` graph



🟧 waiting

🟥 computing/allocated

🟩 memory freed

[lab] Dask arrays

- Let’s run together the notebooks `dask/03-dask-array.ipynb` and `04-dask-array-from-file.ipynb`

F2PY *is a Fortran to Python interface generator. It’s part of NumPy and also is available as a standalone command line tool* `f2py` *that’s installed with NumPy. It facilitates creating and building Python C/API extension modules that provide a connection between Python and Fortran.*

[lab] Python binding with F2PY for a Cityblock distance matrix Fortran90 subroutine

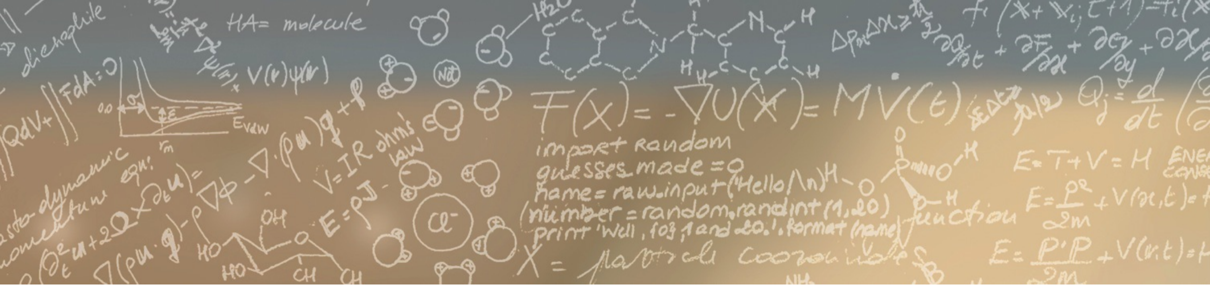
- Let’s go to the notebook `cityblock-distance-matrix-fortran.ipynb`.
  - Open a terminal and build the libraries. The same python executable that’s used for the kernel must be used to build the libraries. You can find the path to it on the kernel’s launcher script.
  - Notice that the empty array is created with `order='F'`. and that the `x` array containing the dataset is passed transposed.
  - You may try to build the libraries using the ‘by hand’ command described on the notebook. Make sure that you copy the `.so` files to the folder `metrics` to the directory level where the notebooks are running. Alternatively you could add the directory that contains the `.so` files to the `PYTHONPATH`.
- Run all cells and compare the execution times of the different approaches.
- While running the `%timeit` function calls, you may open a terminal and check with `top` that the function runs in multiple threads.

CFFI

CFFI *(C Foreign Function Interface for Python) enables calling C code from Python without learning a third language, to be used as interface. CFFI interacts with almost any C code from Python, based on C-like declarations.*

[lab] Python binding with CFFI for a Cityblock distance matrix C function

- Let’s go to the folder `cityblock-cffi`.
  - Open a terminal and build the libraries following the instructions on the `README.md` file. The same python executable that was used for the kernel on the other tutorials must be used to build the libraries. You can find the path to it on the kernel’s launcher script.
  - Notice that in this case two libraries are generated. A C library that contains the Cityblock distance matrix function and a python-importable library that’s linked to it, which does the biding from C to Python.
  - You may create a notebook similar to the one with F2PY where you write a wrapper function for the `cbdm` C function. You can time it then and compare it to the NumPy implementation. While running the `%timeit` function calls, you may open a terminal and check with `top` that the function runs in multiple threads.



Thank you for your attention!

ETH zürich

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre