Distributed Algorithm Dimensionality Reduction

Allegri Luca

De Masi Michele

Grilli Alessandro

Sbarbati Riccardo

Management and Analysis of Physics Dataset - mod. B

Padua University

Physics of Data

A.Y. 2021/2022

Introduction

- Dimensionality Reduction;
- Parallelized SVD;
- Code implementation through Dask;
- Cluster creation;
- Analysis of *California Housing Dataset;*
- HAVOK algorithm code and implementation;
- Analysis of Lorenz System Dataset;

Dimensionality Reduction

Singular Value Decomposition (SVD)

- Decomposition of **A**, *tall and skinny*, matrix using SVD:

$$A = U\Sigma V^T$$

- Implementation of **Direct TSQR** algorithm with a MapReduce architecture;
- The distributed system is built with Dask;
- Stable algorithm for *ill-conditioned* matrices as well;

Direct TSQR
Step 1

```
## Apply numpy QR decomposition to A using Dask map
S1 = A.map(np.linalg.qr)

## Save the Qj matrices into the machine storage using the perisist() function
Q1 = client.persist(S1.map(extract_Q))
```

$$A = \underbrace{\begin{bmatrix} Q_1 & & & \\ & Q_2 & & \\ & & Q_3 & \\ & & & Q_4 \end{bmatrix}}_{8n \times 4n} \underbrace{\begin{bmatrix} R_1 \\ R_2 \\ R_3 \\ R_4 \end{bmatrix}}_{4n \times n}$$

Compute a single QR Decomposition

Output: $\mathbf{Q_{1,j}}$ and $\mathbf{R_{1,j}}$

Direct TSQR
Step 2

$$\underbrace{\begin{bmatrix} R_1 \\ R_2 \\ R_3 \\ R_4 \end{bmatrix}}_{4n \times n} = \underbrace{\begin{bmatrix} Q_1^2 \\ Q_2^2 \\ Q_3^2 \\ Q_4^2 \end{bmatrix}}_{4n \times n} \underbrace{\tilde{R}}_{n \times n}$$

- Compute a single QR Decomposition over $\mathbf{R}_{1,j}$

  Output: **R** and $\mathbf{Q_2}$

- Apply local **SVD** over **R**

  Output: $\mathbf{U_1}$, **S**, $\mathbf{V^T}$

```
#extract R from the QR decomposition and compute it
R1_col = S1.map(extract_R).flatten().compute()
#perform a numpy QR decomposition over the computed R matrix
Q2, R = np.linalg.qr(R1_col)

#compute a numpy SVD over the R matrix
U1, S, Vt = np.linalg.svd(R)
```

Direct TSQR
Step 3

```
#convert Q1 and Q2 into dask array
Q2_p = da.stack(split_rows(Q2, n_partitions, 8), axis=0)
Q1_p = da.stack(list(Q1), axis = 0)

#perform a matrix product
Q = da.map_blocks(lambda a,b : a @ b, Q1_p, Q2_p)

#delete Q1 from the storage
del Q1
```

$$\underbrace{Q}_{8n\times n} = \underbrace{\begin{bmatrix} Q_1 & & & \\ & Q_2 & & \\ & & Q_3 & \\ & & & Q_4 \end{bmatrix}}_{8n\times 4n} \underbrace{\begin{bmatrix} Q_1^2 \\ Q_2^2 \\ Q_3^2 \\ Q_4^2 \end{bmatrix}}_{4n\times n} = \underbrace{\begin{bmatrix} Q_1Q_1^2 \\ Q_2Q_2^2 \\ Q_3Q_3^2 \\ Q_4Q_4^2 \end{bmatrix}}_{8n\times n}$$

- Convert into Dask array

- Matrix Product over $\mathbf{Q}_{1,j}$ and $\mathbf{Q}_2$
  Output: **Q** matrix

Direct TSQR
Step 4

Matrix product between **Q** and $\mathbf{U_1}$

Output: final **U** of the **A**'s SVD

```
#matrix prodoct
U = da.map_blocks(lambda a,b : a @ b, Q, U1)
#computation of U
Compute=True
if Compute == True:
    #concatenate of the U partitioned matricies
    U = np.concatenate(U.compute(), axis=0)
```

Direct TSQR

Final Function

```python
def parallel_SVD(A, n_partitions, Compute=True, Accuracy=False):

    #First step
    S1 = A.map(np.linalg.qr)
    Q1 = client.persist(S1.map(extract_Q))

    #Second step
    R1_col = S1.map(extract_R).flatten().compute()
    Q2, R = np.linalg.qr(R1_col)

    #Second step for SVD
    U1, S, Vt = np.linalg.svd(R)

    #Third step
    Q2_p = da.stack(split_rows(Q2, n_partitions, len(R)), axis=0)
    Q1_p = da.stack(list(Q1), axis = 0)
    Q = da.map_blocks(lambda a,b : a @ b, Q1_p, Q2_p)

    del Q1

    #Fourth step (SVD only)
    U = da.map_blocks(lambda a,b : a @ b, Q, U1)

    if Compute == True:
        U = np.concatenate(U.compute(), axis=0)
    if Accuracy == True:
        print("Accuracy Q: ",est(np.concatenate(Q.compute(), axis=0)))
        print("Accuracy U: ",est(U))

    return(U,S,Vt)
```

Clusters Implemented

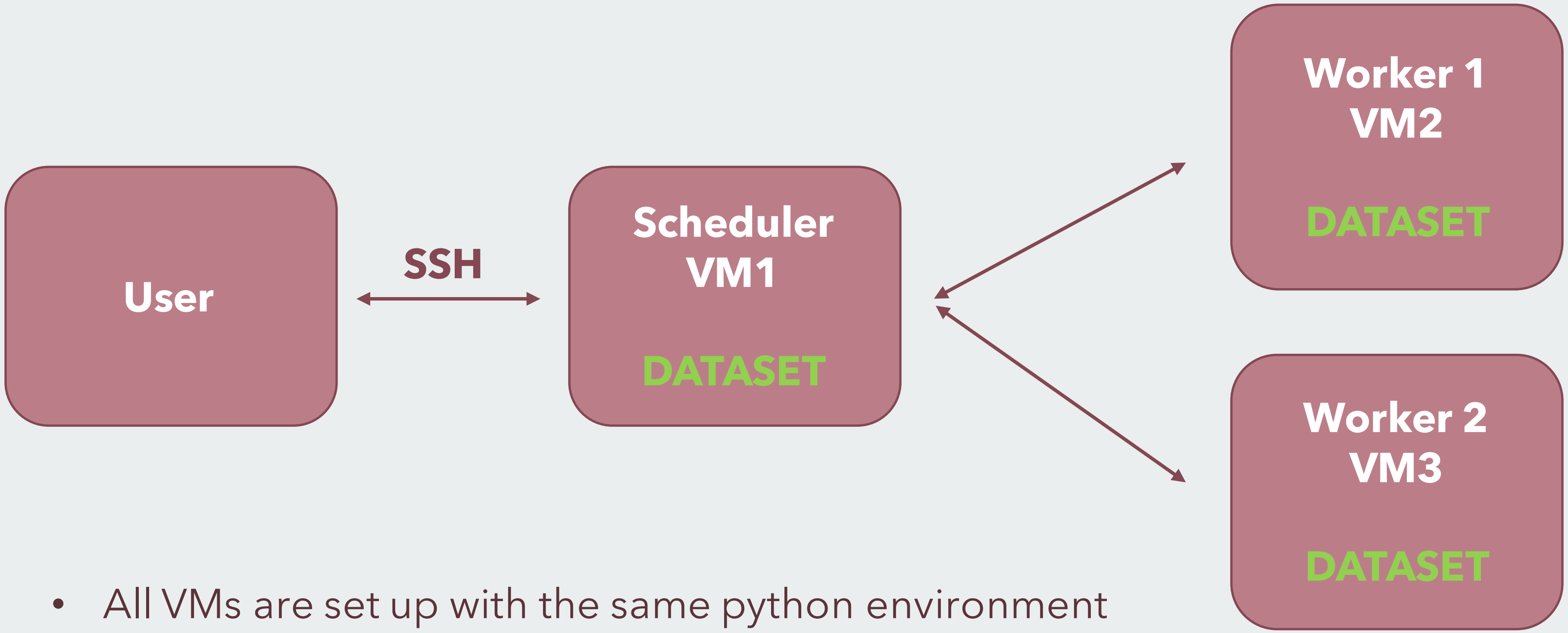
- Code development: Local cluster built from Docker image.
- Running and performance tests: Remote cluster of Virtual Machines (CloudVeneto instances).

Each instance of the remote cluster is equipped with:

- 4 cores with an Intel(R) Xeon(R) CPU @ 2.50 GHz.
- 7.8 GB RAM.

The total RAM is evenly split among the worker cores defined for each instance (max. 4).

Remote cluster



- All VMs are set up with the same python environment and all authentication requirements are removed.
- The mount folder **DATASET** contains all the data.

First analysis – California Housing dataset

From `sklearn.datasets`.

20640 instances, each described by the 8 following features:

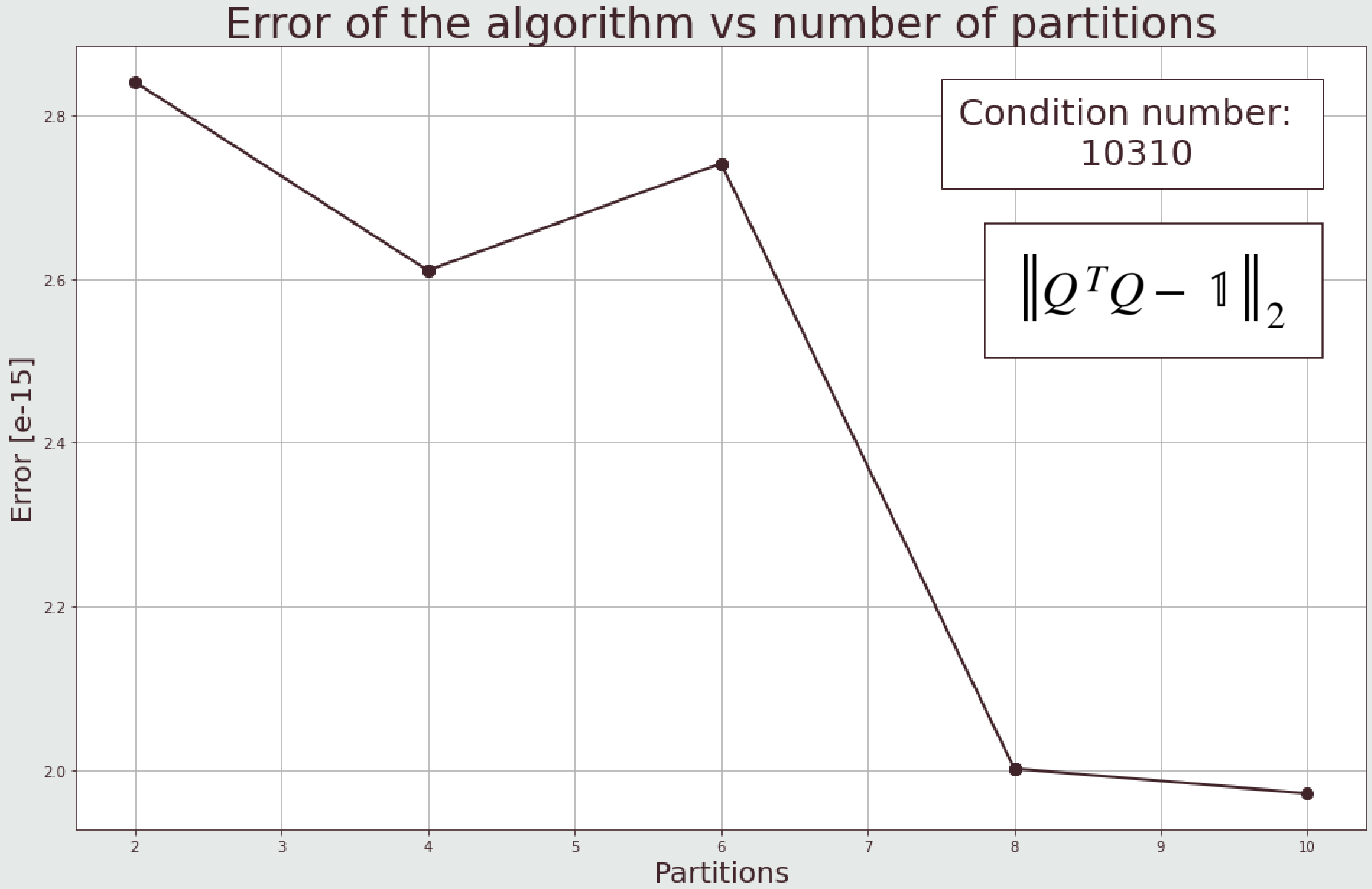
- *MedInc*: median income in block group
- *HouseAge*: median house age in block group
- *AveRooms*: average number of rooms per household
- *AveBedrms*: average number of bedrooms per household
- *Population*: block group population
- *AveOccup*: average number of household members
- *Latitude*: block group latitude
- *Longitude*: block group longitude

The target variable is the median house value for California districts.

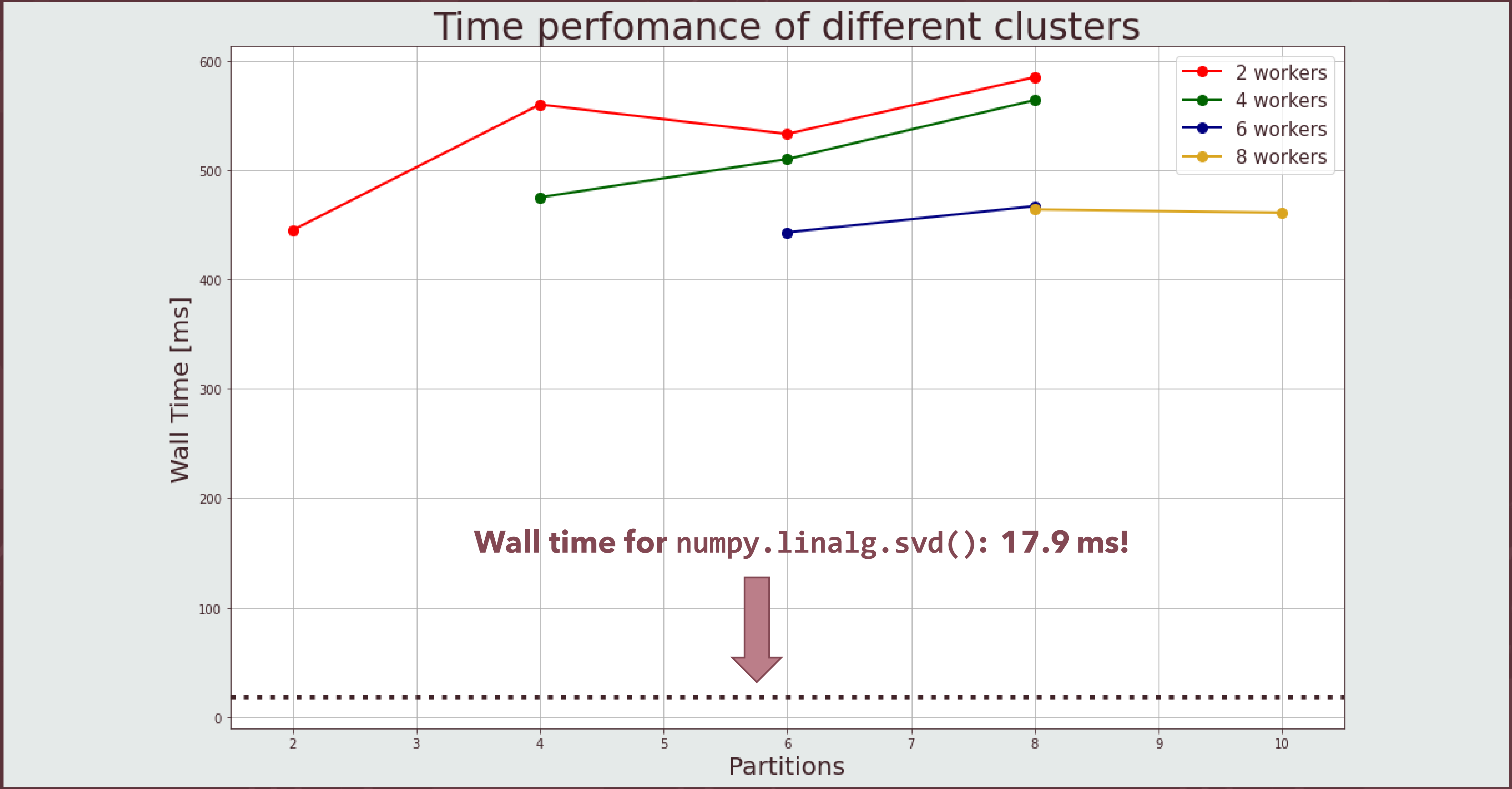
Data preprocessing

- Depending on the number of partitions desired, sections of the dataset are saved into *.json* files.
- The data in each file is loaded in a section of a *Dask bag* called `X_b`.
- The data in the bag are in a dictionary-like structure, so they have to be converted into matrices.
- The resulting object `A` is now ready to be fed to `Parallel_SVD()`.

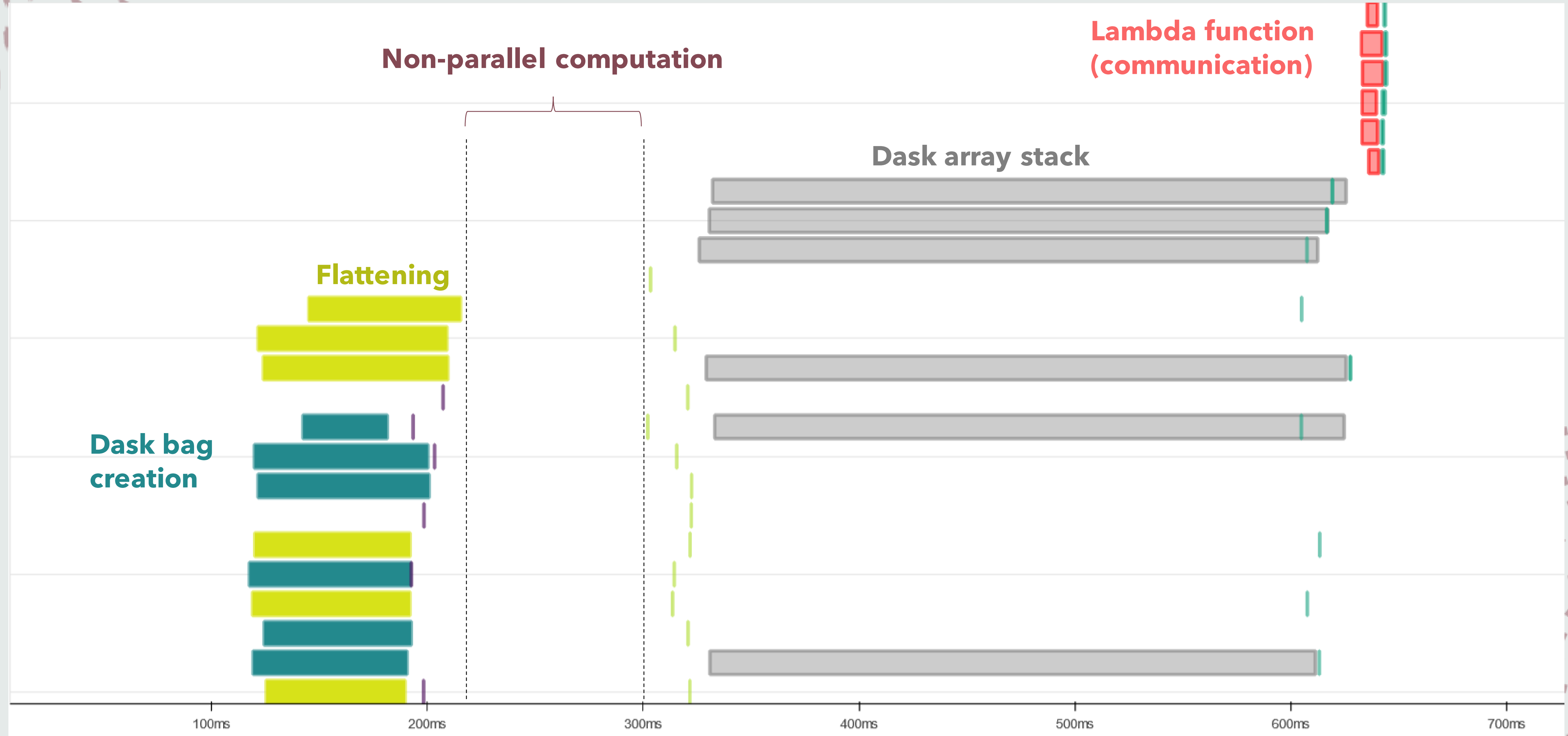
```
for i in range(n_partitions):
    X[i*int(len(X)/n_partitions):(i+1)*int(len(X)/n_partitions)].to_json('DATASET/California/X.{}.json'.format(i+1))

X_b = db.read_text(os.path.join('DATASET','California','X.*.json')).map(json.loads)

A = X_b.map(A_matrix)
```


Error of the algorithm vs number of partitions
Condition number:
10310
$\left\| Q^T Q - \mathbb{1} \right\|_2$
Error [e-15]
2.0
2.2
2.4
2.6
2.8
2
3
4
5
6
7
8
9
10
Partitions


Time perfomance of different clusters
Wall Time [ms]
600
575
550
525
500
475
450
425
400
2
3
4
5
6
7
8
9
10
Partitions
2 workers
4 workers
6 workers
8 workers


Time perfomance of different clusters
Wall Time [ms]
Partitions
600
500
400
300
200
100
0
2
3
4
5
6
7
8
9
10
2 workers
4 workers
6 workers
8 workers
Wall time for numpy.linalg.svd(): 17.9 ms!

Task stream


Lambda function
(communication)
Non-parallel computation
Dask array stack
Flattening
Dask bag
creation
100ms
200ms
300ms
400ms
500ms
600ms
700ms

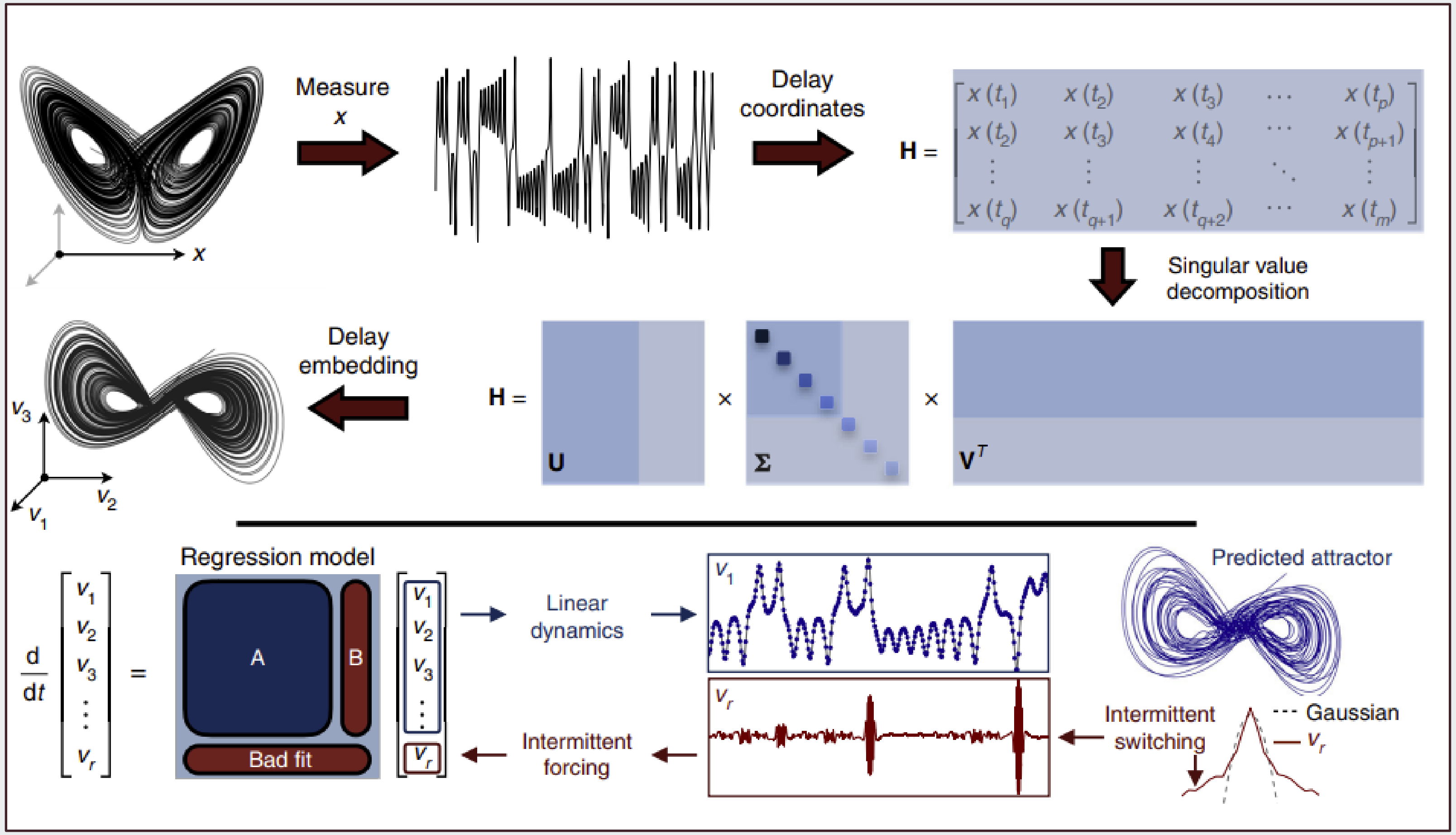
HAVOK - Hankel Alternative View Of Koopman

HAVOK consists of a universal, data-driven decomposition of chaos as an intermittently forced linear system, performed with a combination of delay embedding and Koopman theory.

The algorithm is able to build a linear representation of a multi-dimensional chaotic system, starting from a unidimensional time series of the most meaningful coordinate.

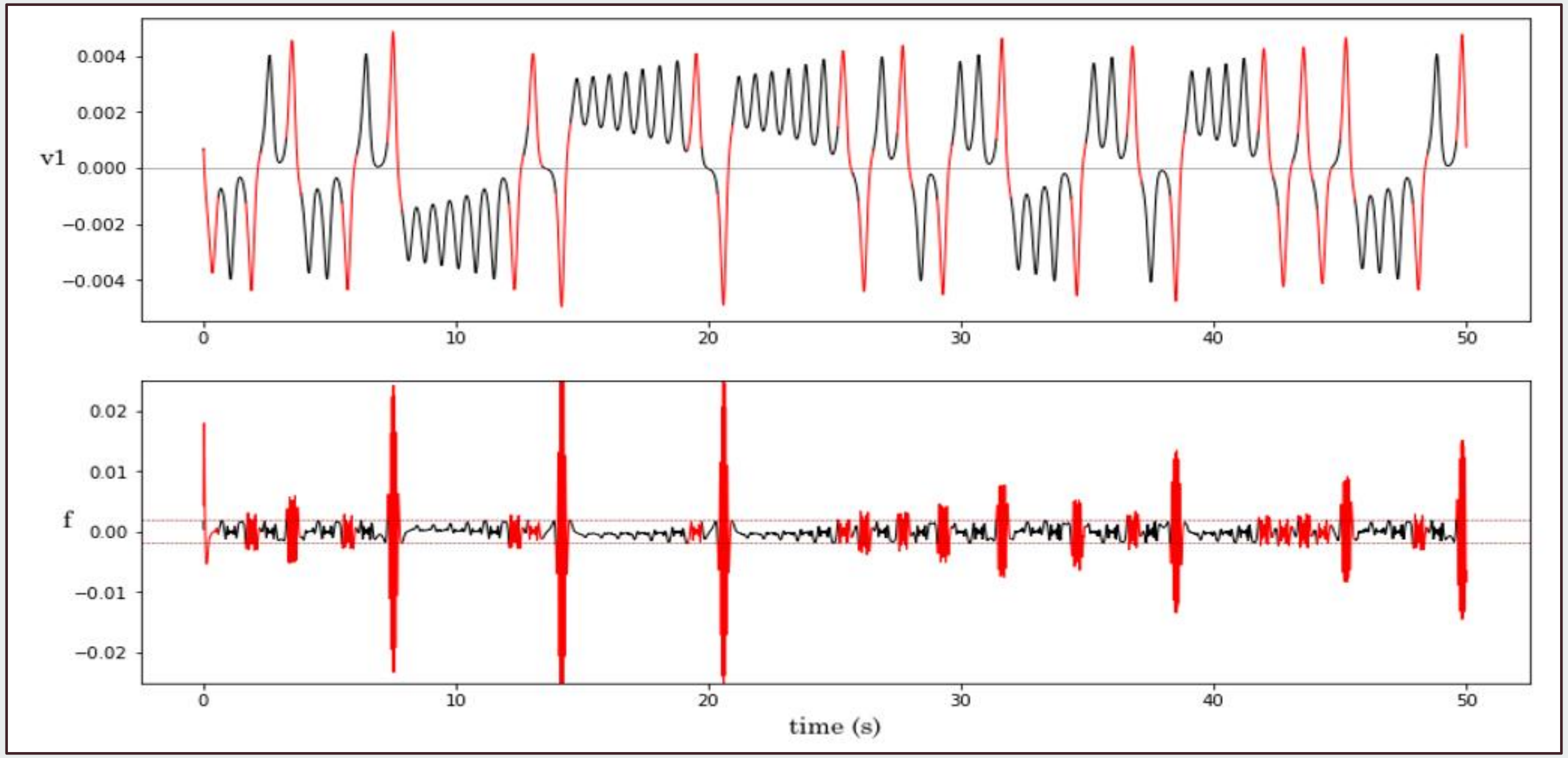
In this work, HAVOK analysis is performed on the Lorenz attractor, a well known example of chaotic dynamics.

HAVOK - Hankel Alternative View Of Koopman



Lobe switching prediction

By setting a threshold in the forcing term it is possible to predict shortly in advance the chaotic behaviour of the Lorenz system *i.e.* the lobe switching.



Distributed HAVOK analysis of a Lorenz system

The first step of the HAVOK analysis consists of the division of the time series into windows of length `l` and the creation of a Hankel matrix `H`.

For the distributed case it is created a Hankel matrix for every chunk with the condition that the components of each one are smoothly connected.

```
def Hankel(x_ts,l):
    l=100
    H = np.zeros((len(x_ts),l))
    for i in range(len(x_ts)-l): H[i,:] = x_ts[i:l+i]
    return H

l = 100
H = x_b.map(lambda x: Hankel(x,l))
```

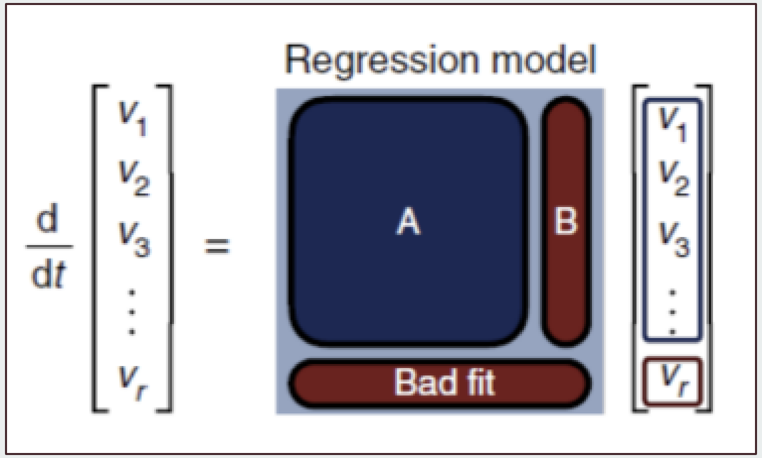
Hankel matrix Parallel SVD

`H` is a "tall-and-skinny" matrix so we can perform the parallel SVD through *Direct TSQR*. In this case `U` is a matrix with the same dimensions of `H` but to perform the HAVOK analysis only the first `r` columns are needed.

```
U,S,Vt = parallel_SVD(H, n_partitions=n_train, Compute=False)

r = 15
u = np.concatenate(np.array(U[:,:,:r]),axis=0)
```

Sparse regression and integration of the linear system

- Local sparse regression with pySindy
- Parallel integration of the linear model
- Comparison with the embedded time series



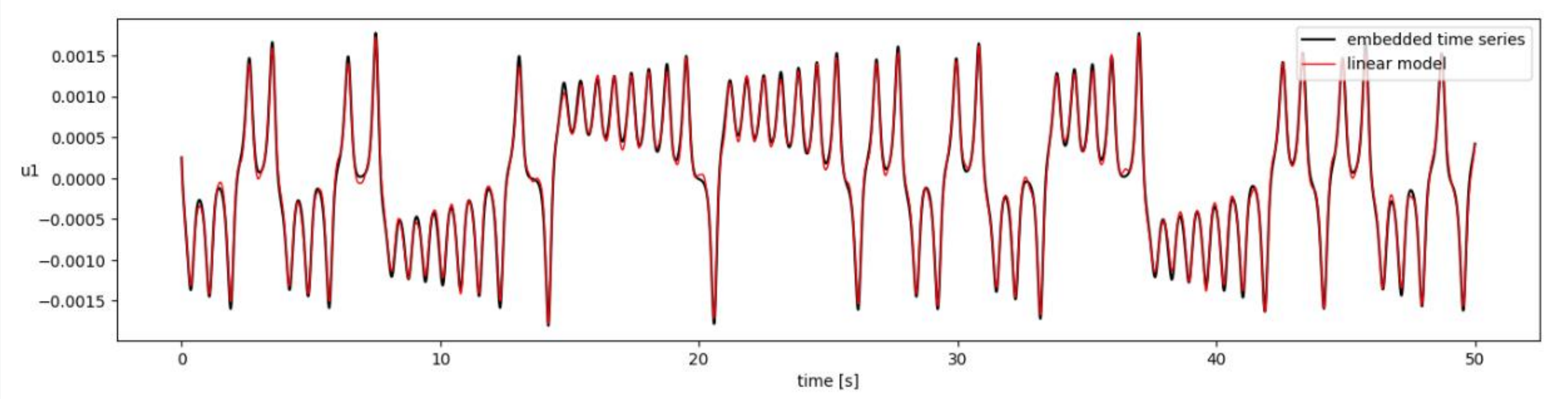
Sparse regression and integration of the linear system

- Local sparse regression with pySindy
- Parallel integration of the linear model
- Comparison with the embedded time series

```
sys = control.StateSpace(A,B,np.eye(r-1),0*B)
Ur = db.from_sequence(np.array(U[:,:,:r]))

# the function lsim return something of the kind (y_out,time,...)
def extract_Y(M):
    Y = M[:][0]
    return Y

result = Ur.map(lambda x: control.matlab.lsim(sys,x[:,r-1],np.arange(0,len(x)*dt,dt),x[0,:r-1]))\
            .map(extract_Y).compute()

y = np.concatenate(result,axis=0)
```

Sparse regression and integration of the linear system

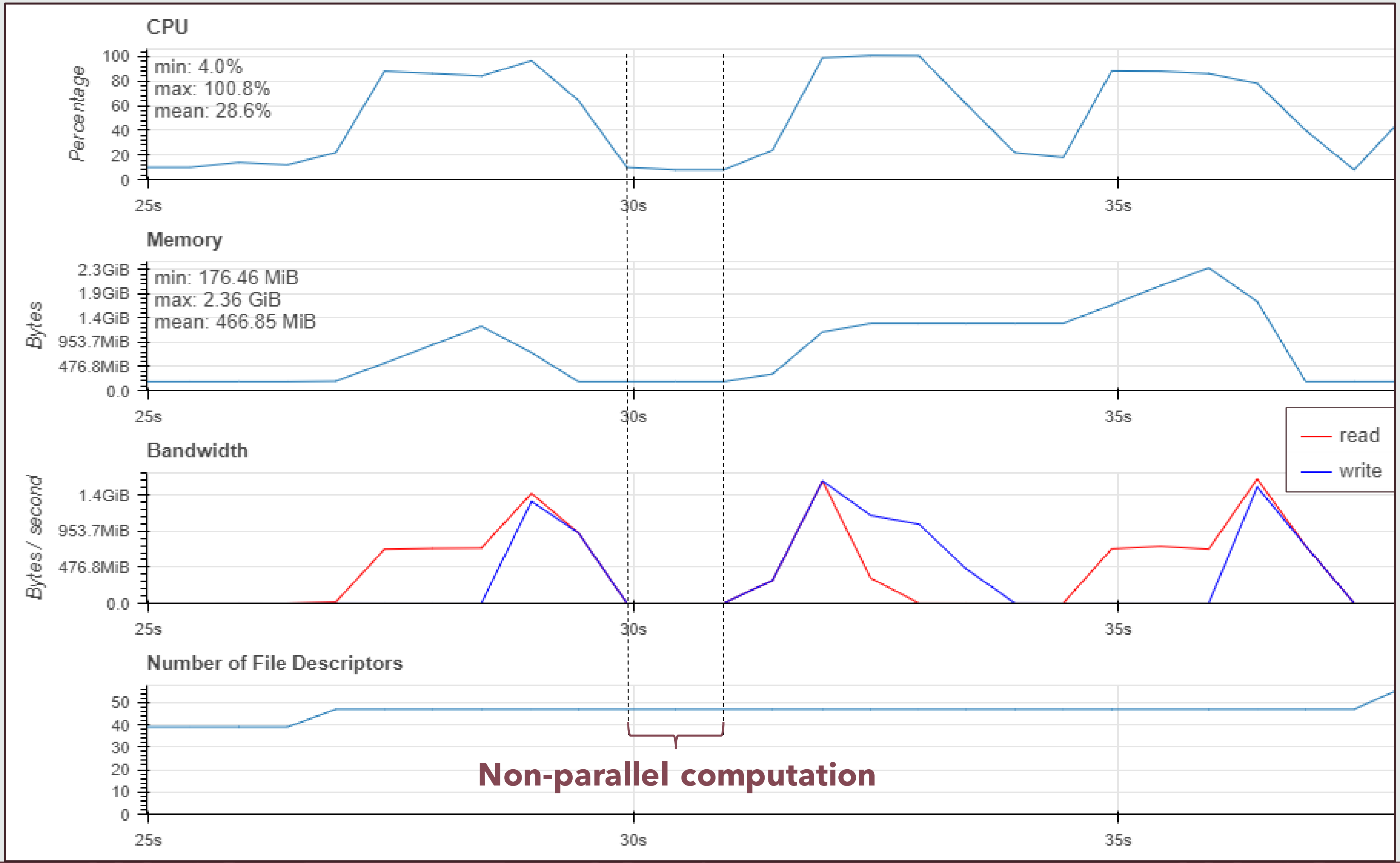
- Local sparse regression with pySindy
- Parallel integration of the linear model
- Comparison with the embedded time series

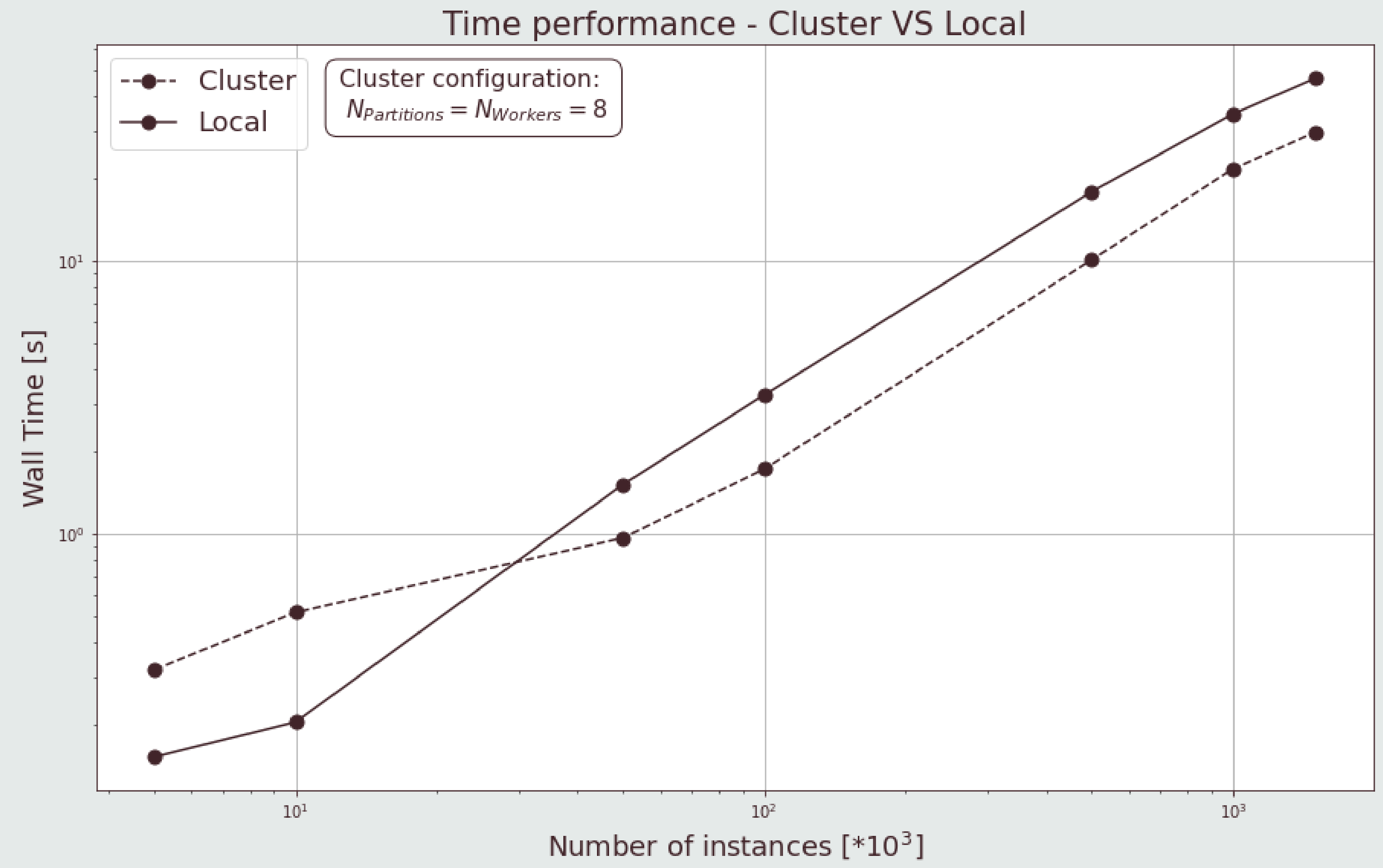



Time perfomance of different clusters
Standard SVD
2 workers
4 workers
6 workers
8 workers
Wall Time [s]
Partitions
55
50
45
40
35
30
25
2
3
4
5
6
7
8
9
10


Time perfomance of different clusters
Standard SVD
2 workers
4 workers
6 workers
8 workers
Wall Time [s]
Partitions
55
50
45
40
35
30
25
2
3
4
5
6
7
8
9
10
4 is faster than 6!

System performance for the best cluster




Time performance - Cluster VS Local
Cluster
Local
Cluster configuration:
$N_{Partitions} = N_{Workers} = 8$
Wall Time [s]
10^1
10^0
10^1
10^2
10^3
Number of instances [$*10^3$]

Conclusions

- Good accuracy in the decomposition of "tall-and-skinny" matrices
- Better time performances (than non-distributed SVD) for the right number of workers and partitions
- Convenient for big enough dataset