Journée Cascimodot 10/12/2021

Implémentation d’algorithmes en Python : Retour d’expérience sur l’utilisation du cluster mirev (LIFO)

Agenda

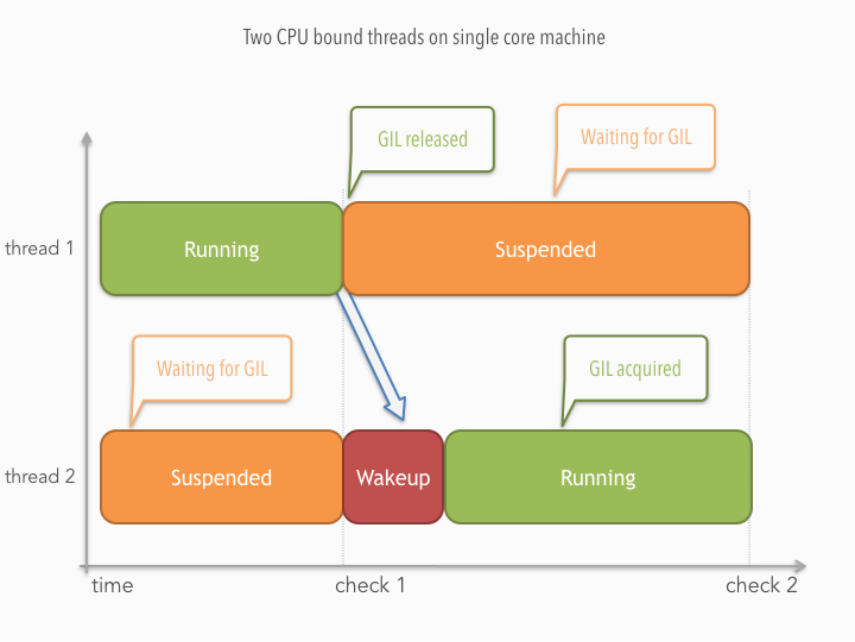
- Méthodologie pour algorithmes efficaces
- Les différentes options « simples » :
  - Numpy
  - Joblib
  - Numba
- Comparaison des performances
- Où j’utilise ces méthodes

Méthodologie pour algorithmes efficacies

- Pourquoi Python ?

Les points d'attention concernant le parallélisme:

- Identifier et isoler les taches indépendantes dans des fonctions
- Identifier les endroits où le GIL Python n'est pas necessaire



De manière générale :

- Minimiser le nombre de paramètres en entrée
- Identifier lorsqu'on peut imposer une limite sur les types de données (float8 plutôt que float64, uint plutôt que int)
- Privilégié les opérations "in-place" (a += b plutôt que a = a + b)
- Mettre en cache les résultats plutôt que recalculer lorsque c'est possible/raisonnable

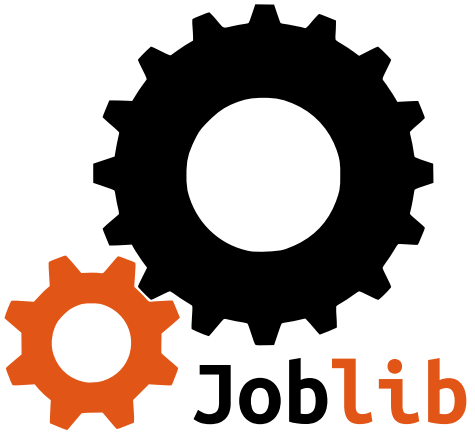
Les différentes options : Numpy

Avantages	Inconvénients
Implémentation efficace en C	**Aucun parallélisme**
Vectorisation via Single instruction, multiple data (SIMD)	*Allocation contiguë des tableaux en mémoire*
Couvre la majorité des opérations mathématiques	Peu/Pas d'avantages sur les petits tableaux (<100)

- Très bonne performance sur une majorité de problemes.
- Demande un haut niveau de connaissance de la librairie et de ses mechanismes pour obtenir les meilleures performances.

Les différentes options : Joblib

Avantages	Inconvénients
Parallélisation de n'importe quelle fonction et type de donnée	Utilise des processus plutôt que des threads par défaut
Offre des options de cache sur disque / persistance des objets	
Préserve l'ordre des operations dans le résultat	



- Crée un ensemble de workers (process/threads) et affecte les operations via un itérateur.
- Beaucoup de coûts supplémentaires (copie des données/fonctions) lors de l'utilisation des processus.
- On utilise des processus quand notre code ne peut pas relacher le GIL, des threads dans le cas contraire.
- S'interface facilement avec *tqdm* pour afficher des barres de progression

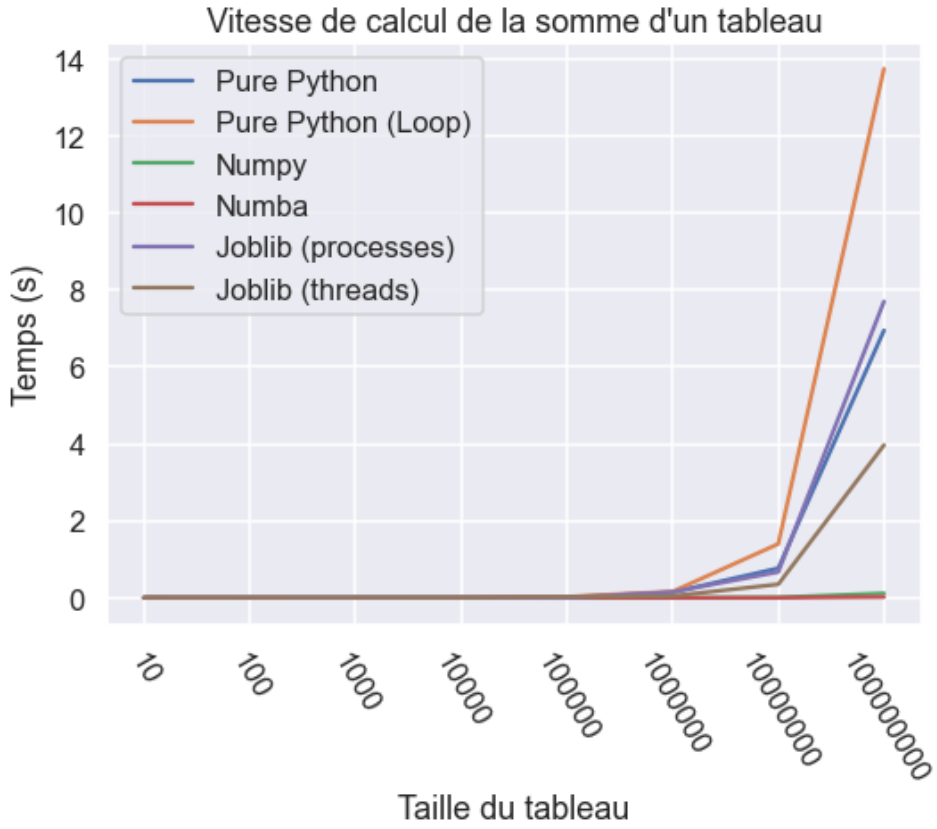
```
>>> from joblib import Parallel, delayed
>>> from math import sqrt
>>> Parallel(n_jobs=1)(delayed(sqrt)(i**2) for i in range(10))
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
```

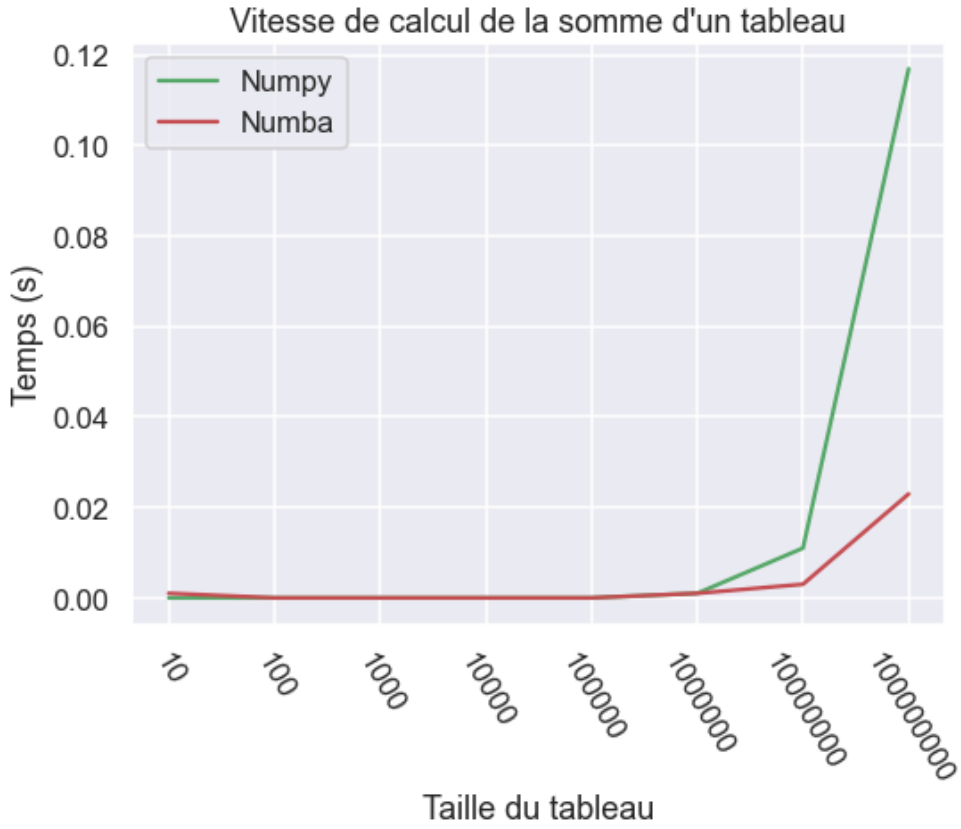
Les différentes options : Numba

Avantages	Inconvénients
JIT compilation, SIMD vectorization, Parallélisation, Support GPU	Restreint à des entrées via tableau/type numpy
Possibilité de “compiler” les fonctions python en C et de définir des fonctions sans GIL	Très sensible au type de donnée (e.g pas de convertion automatique sur des opérations entre float32 et float64)
Nombreux outils et options pour optimiser le code. Supporte beaucoup d’opération de Numpy	

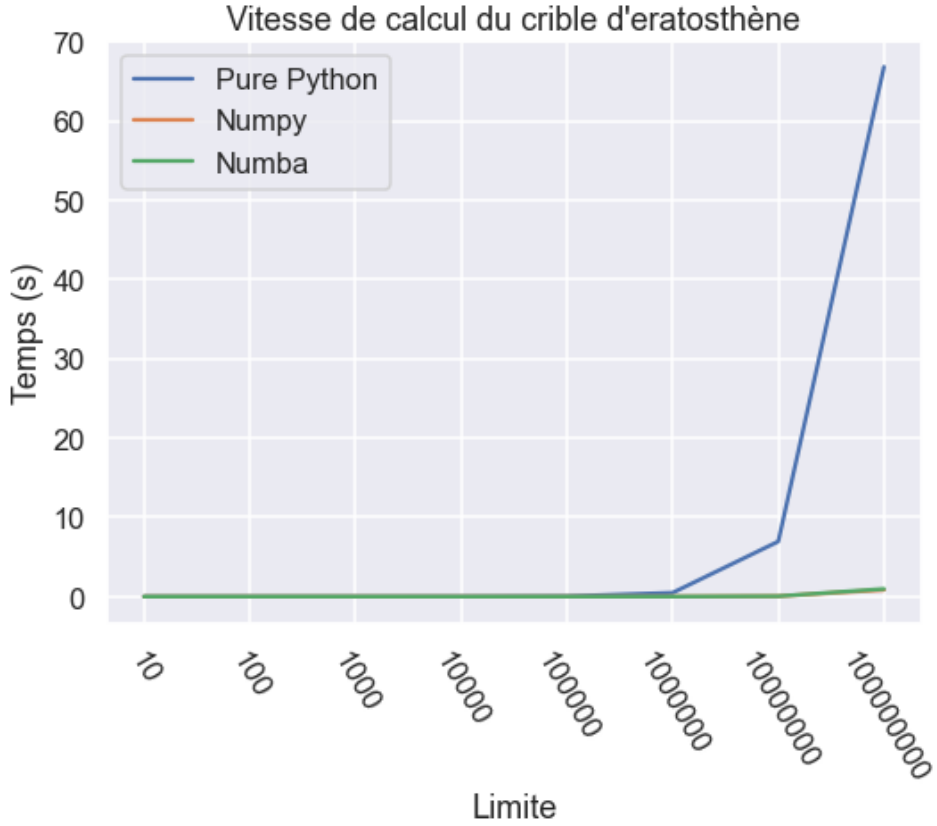
```python
@njit(parallel=True)
def do_sum_parallel(A):
    # each thread can accumulate its own partial sum, and then a cross
    # thread reduction is performed to obtain the result to return
    n = len(A)
    acc = 0.
    for i in prange(n):
        acc += np.sqrt(A[i])
    return acc
```

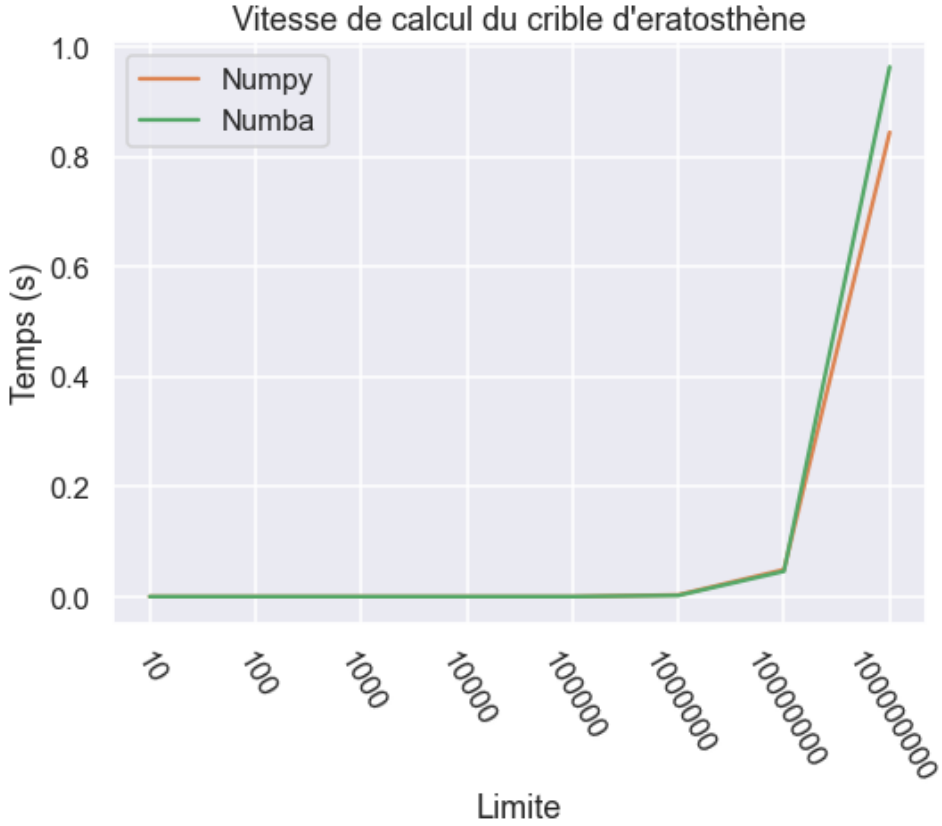
Comparaison des performances





Comparaison des performances





Où j'utilise ces méthodes

Dans mes travaux sur les algorithmes de transformation et de classification des données temporelles.

Par exemple pour un algorithme utilisant un grand nombre de noyaux de convolution aléatoire (>10000).

@njit = @jit(nogil=True)

Depuis $X = x_0, \dots, x_n$, la longueur et la dilation d'un noyau, on veut $C = c_0, \dots, c_{n-(l-1*d)}$, où $c_i = f([x_i,\ x_{i+d}, \dots, x_{i+(l-1)*d}])$

```python
@njit(cache=True)
def generate_strides_1D(X, window_size, dilation):
    n_timestamps = X.shape[0]

    shape_new = (n_timestamps - (window_size-1)*dilation,
                 window_size)

    s0 = X.strides[0]
    strides_new = (s0, dilation *s0)
    return as_strided(X, shape=shape_new, strides=strides_new)

@njit(cache=True)
def convolve_x(x, length, dilation, padding):
    n_timestamps = x.shape[0]
    if padding > 0:
        x_pad = np.zeros(n_timestamps + 2 * padding)
        x_pad[padding:-padding] = x
    else:
        x_pad = x
    return generate_strides_1D(x_pad, length, dilation)
```

Où j'utilise ces méthodes

On extrait ensuite de $C = c_0, \ldots, c_{n-(l-1*d)}$ plusieurs statistiques pour obtenir un ensemble de feature. Avec N series temporelles et K noyau, on veut obtenir un tableau (N, 3K).

Sans numba 51.3 s ± 7.51 s
Avec numba : 1.78 s ± 112 ms

Sur un petit ensemble de donnée (1500,1,150) et avec seulement 8 coeurs.


```python
@njit(fastmath=True, cache=True)
def apply_one_kernel_one_sample(x, values, length, dilation, padding):
    x_conv = np.sum(
        (convolve_x(x, length, dilation, padding) - values[:length])**2,
        axis=-1
    )
    return np.min(x_conv), np.mean(np.diff(x_conv) > 0), np.float64(np.argmin(x_c


@njit(cache=True, parallel=True)
def apply_all_kernels(X, values, lengths, dilations, paddings):
    n_samples, n_ft, n_timestamps = X.shape
    n_shapelets = lengths.size
    n_features = 3
    X_new = np.empty((n_samples, n_features * n_shapelets))
    for i in prange(n_samples):
        for j in prange(n_shapelets):
            X_new[i, (n_features * j):(n_features * j + n_features)] = apply_one_k
                X[i,0], values[j], lengths[j], dilations[j], paddings[j]
            )
    return X_new
```

Sources & References

- https://wiki.python.org/moin/ParallelProcessing : Une reference complete de tout les packages JIT, parallele etc…
- https://chelseatroy.com/2018/11/07/code-mechanic-numpy-vectorization/ : Une analyse en profondeur de la performance des raisons de la performance de Numpy
- https://joblib.readthedocs.io/en/latest/parallel.html : Doc Joblib
- https://numpy.org/ : Doc Numpy
- https://numba.pydata.org/numba-doc/latest/index.html : Doc Numba
- https://github.com/baraline/convst : Package convst (WIP) de l’algorithme présenté.