Laboratorio 01
Rendimiento (Comparación) con Joblib

Rendimiento (Comparación)

Calculemos raíces cuadradas de varios números de distintas maneras

```
[ ]  import numpy as np
     import time as tm
     from math import sqrt

     n = 10000000
```

Python

```
[ ]  print("Comenzando a calcular...")
     start = tm.time()
     normal_results = [sqrt(i) for i in range(n)]
     end = tm.time()
     print(f"Tiempo total: {end - start}s")
```

```
Comenzando a calcular...
Tiempo total: 3.9920427799224854s
```

Numpy

```
print("Comenzando a calcular...")
start = tm.time()
data = np.arange(n)
numpy_results = np.sqrt(data)

end = tm.time()
print(f"Tiempo total: {end - start}s")
```

```
Comenzando a calcular...
Tiempo total: 0.3026847839355469s
```

Joblib

```
[ ]  from joblib import Parallel
     from joblib import delayed
```

Dos trabajadores

```
[ ]  print("Comenzando a calcular...")
     start = tm.time()
     parallel_pool = Parallel(n_jobs=2)
     parallel_sqrt = delayed(sqrt)
     parallel_tasks = [parallel_sqrt(i) for i in range(n)]
     parallel_results = parallel_pool(parallel_tasks)
     end = tm.time()
     print(f"Tiempo total: {end - start}s")
```

```
Comenzando a calcular...
Tiempo total: 54.573294162750244s
```

Cuatro trabajadores

```
[ ]  print("Comenzando a calcular...")
     start = tm.time()
     parallel_pool = Parallel(n_jobs=4)
     parallel_sqrt = delayed(sqrt)
     parallel_tasks = [parallel_sqrt(i) for i in range(n)]
     parallel_results = parallel_pool(parallel_tasks)
     end = tm.time()
     print(f"Tiempo total: {end - start}s")
```

```
Comenzando a calcular...
Tiempo total: 43.94699048995972s
```

¿Qué ocurre si usamos la función raíz de Numpy?

```
[ ]  print("Comenzando a calcular...")
     start = tm.time()
     parallel_pool = Parallel(n_jobs=2)
     parallel_sqrt = delayed(np.sqrt) # Notar la diferencia
     parallel_tasks = [parallel_sqrt(i) for i in range(n)]
     parallel_results = parallel_pool(parallel_tasks)
     end = tm.time()
     print(f"Tiempo total: {end - start}s")
```

```
Comenzando a calcular...
Tiempo total: 87.53696823120117s
```

Finalmente con batch_size fijo

```
print("Comenzando a calcular...")
start = tm.time()
parallel_pool = Parallel(n_jobs=2, batch_size=100000)
parallel_sqrt = delayed(sqrt)
parallel_tasks = [parallel_sqrt(i) for i in range(n)]
parallel_results = parallel_pool(parallel_tasks)
end = tm.time()
print(f"Tiempo total: {end - start}s")
```

```
Comenzando a calcular...
Tiempo total: 38.662869453430176s
```

```
print("Comenzando a calcular...")
start = tm.time()
parallel_pool = Parallel(n_jobs=2, batch_size=500000)
parallel_sqrt = delayed(sqrt)
parallel_tasks = [parallel_sqrt(i) for i in range(n)]
parallel_results = parallel_pool(parallel_tasks)
end = tm.time()
print(f"Tiempo total: {end - start}s")
```

```
Comenzando a calcular...
Tiempo total: 45.86530303955078s
```

```
print("Comenzando a calcular...")
start = tm.time()
parallel_pool = Parallel(n_jobs=4, batch_size=int(n/4))
parallel_sqrt = delayed(sqrt)
parallel_tasks = [parallel_sqrt(i) for i in range(n)]
parallel_results = parallel_pool(parallel_tasks)
end = tm.time()
print(f"Tiempo total: {end - start}s")
```

```
Comenzando a calcular...
Tiempo total: 50.74200177192688s
```

Comparación rendimiento de valores de una lista vs un Numpy array con append

```
lista = []
t1 = tm.time()
for i in range(n):
    lista.append(i)
t2 = tm.time()
print(t2 - t1)
```

```
2.0541718006134033
```


```
array = np.array([])
t1 = tm.time()
for i in range(n):
    array = np.append(array, i)
t2 = tm.time()
print(t2 - t1)
```

En resumen:

- Si necesitan ir agregando valores -> Listas
- Para realizar operaciones matriciales y de vectores -> Numpy

** Es común agregar valores a una lista y luego transformarla a un array para realizar operaciones con ella.