Calcolo Parallelo e Distribuito

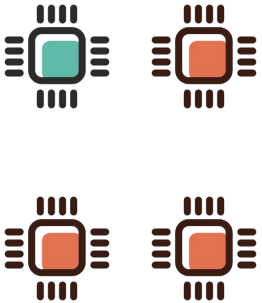
Parte 1

Cezar Sas
c.a.sas@rug.nl

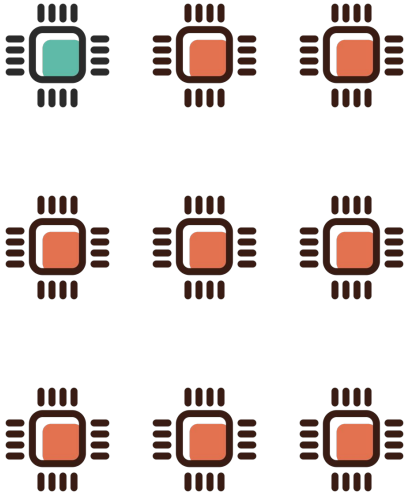
Obiettivi

- Avere un'idea delle varie tipologie di parallelizzazione
- Scrivere un semplice programma che sfrutti più processori
- Risparmiare tempo

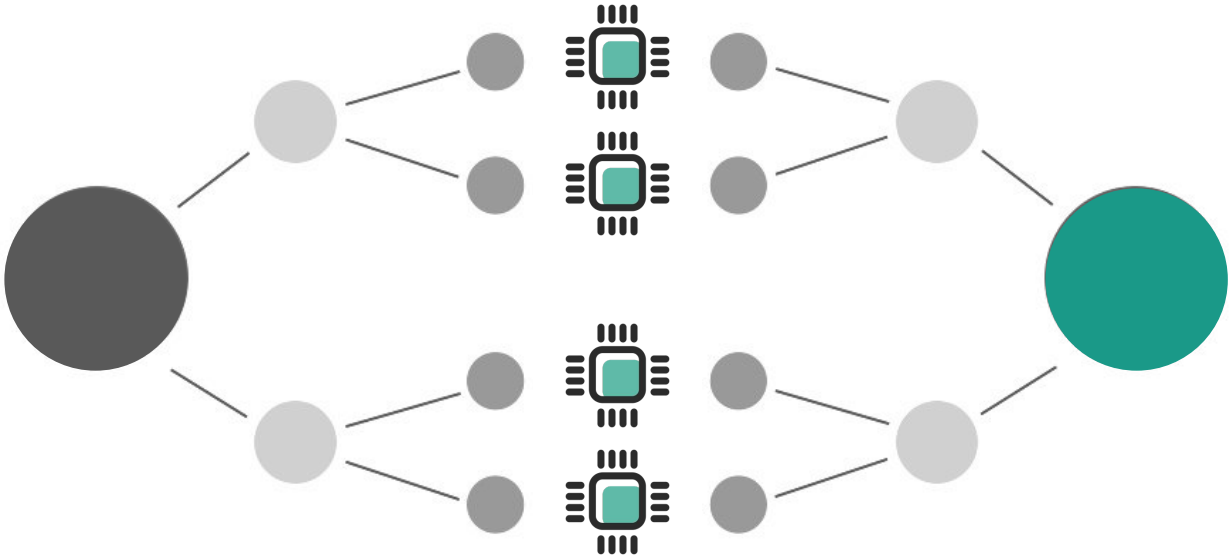
Motivazioni



Motivazioni



Divide et Impera

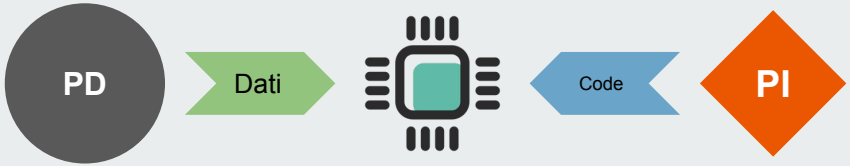


Tassonomia di Flynn

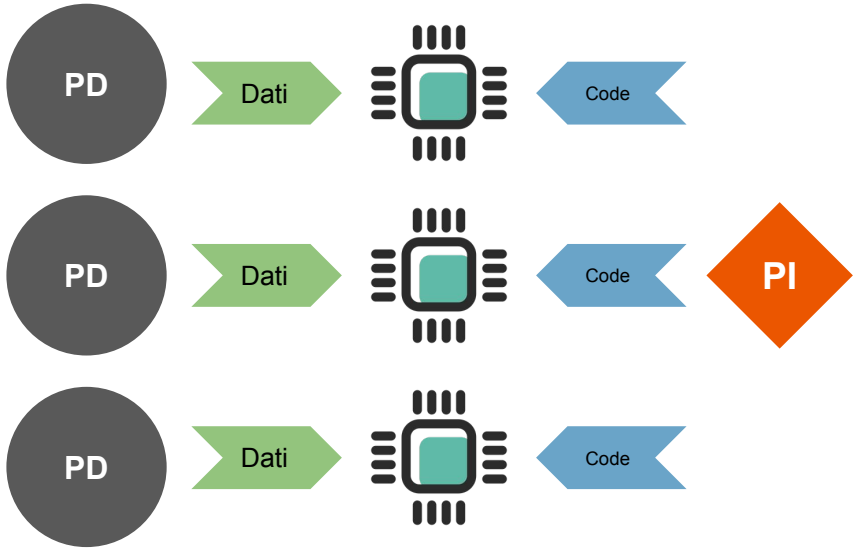
Reference

Architetture 1/2

SIMD
Single Instruction
Multiple Data

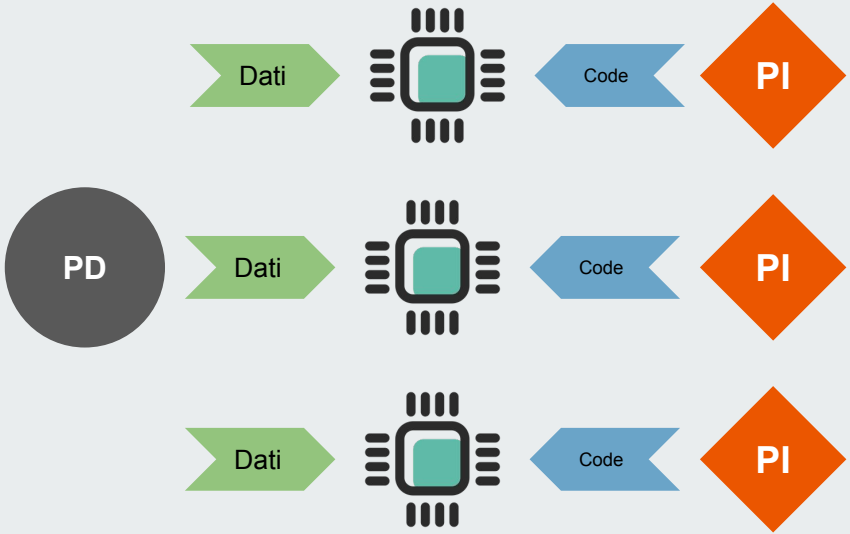


SIMD
Single Instruction
Multiple Data

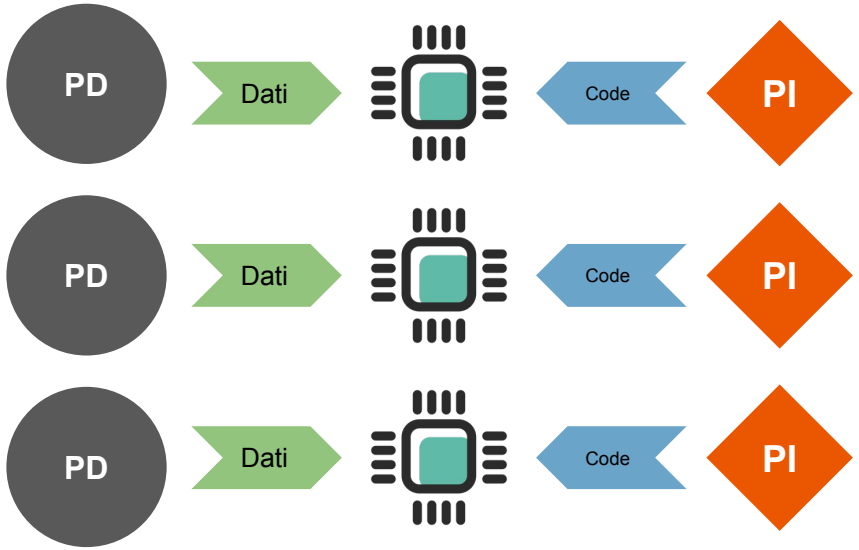


Architetture 2/2

MISD
Multiple Instruction
Single Data

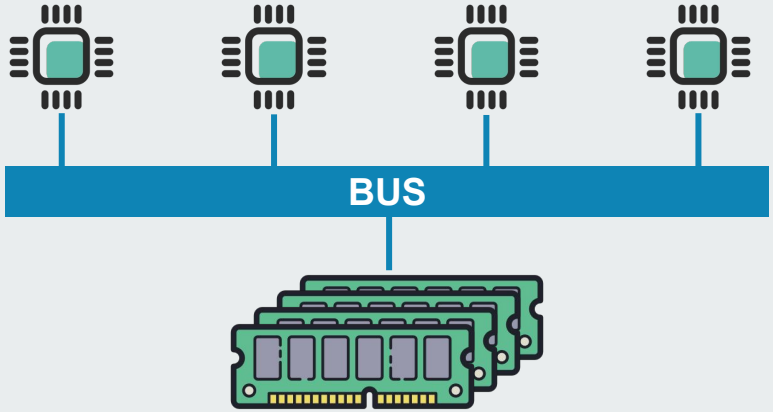


MIMD
Multiple Instruction
Multiple Data



MIMD

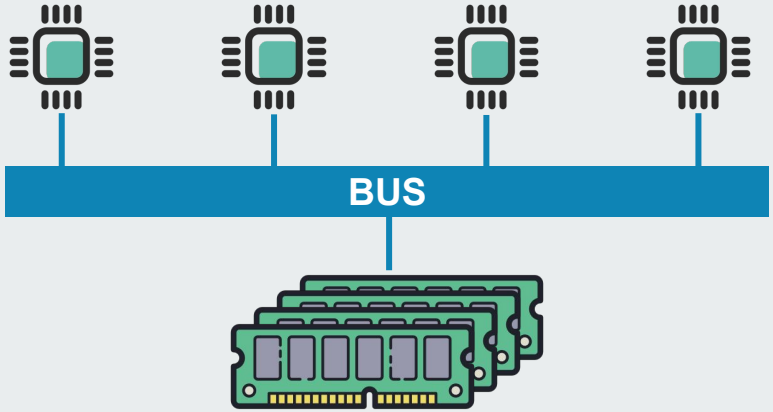
Memoria Condivisa



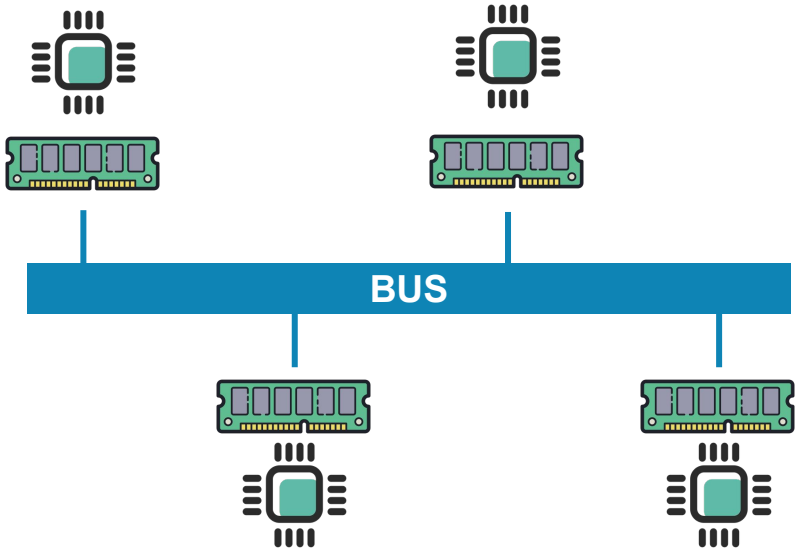
Memoria Distribuita

MIMD

Memoria Condivisa



Memoria Distribuita



Librerie

Oggi

- Multi**Threading**
- Multi**Processing**
- JobLib

Prossime Lezioni

- Ray
- Ray
- Ray

MultiThreading

- Memoria Condivisa
- Ideale per Task I/O Bound
- Thread Veri
- Soggetto a **GIL**

MultiProcessing

- Memoria Separata (< 3.8)
- Memoria Condivisa (> 3.8)
- Maggiore Memoria
- Bypassa il **GIL**

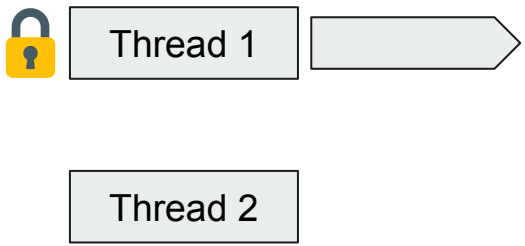
Global
Interpreter
Lock

GIL

```
static PyThread_type_lock interpreter_lock = 0; /* This is the GIL */
```

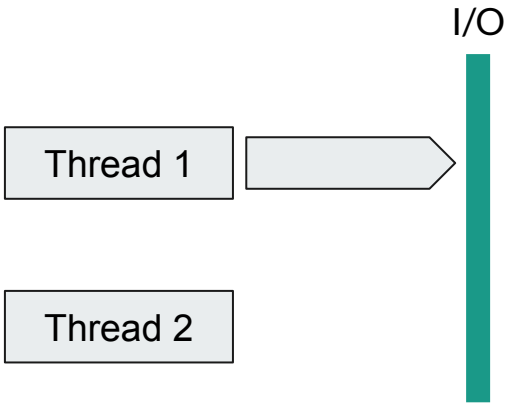
GIL

```
static PyThread_type_lock interpreter_lock = 0; /* This is the GIL */
```



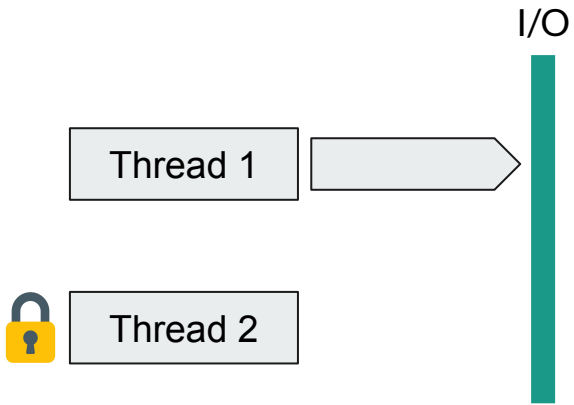
GIL

```
static PyThread_type_lock interpreter_lock = 0; /* This is the GIL */
```



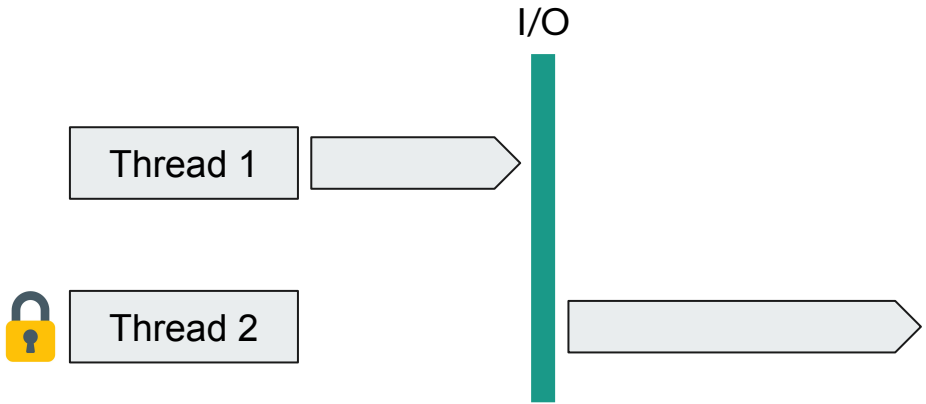
GIL

```
static PyThread_type_lock interpreter_lock = 0; /* This is the GIL */
```



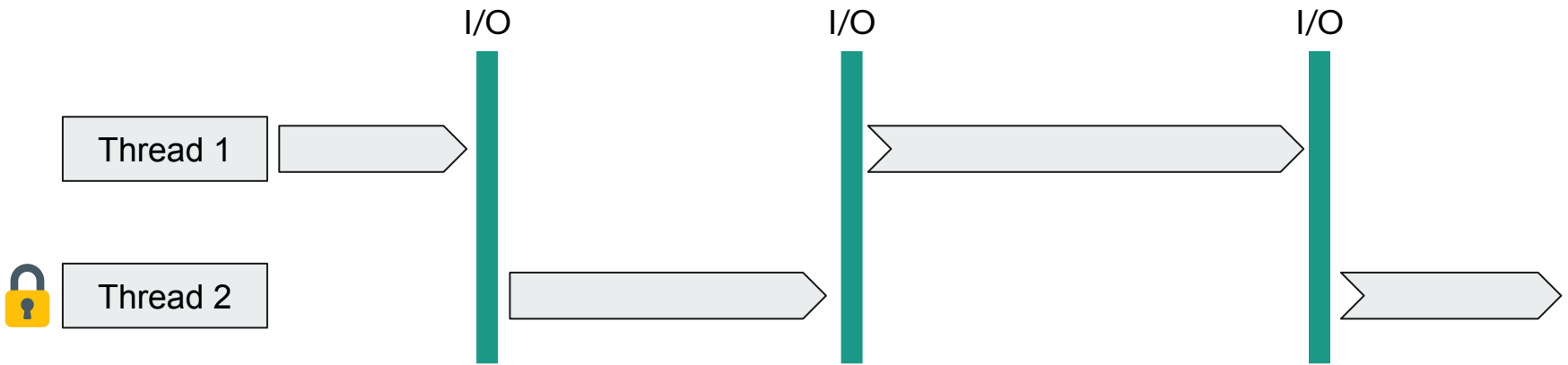
GIL

```
static PyThread_type_lock interpreter_lock = 0; /* This is the GIL */
```



GIL

```
static PyThread_type_lock interpreter_lock = 0; /* This is the GIL */
```



MultiThreading

Utilizzo

```
import threading

t   = threading.Thread(target=func, args=[x, y])
t.start()
t.join()
```

```
import concurrent.futures as cf

with cf.ThreadPoolExecutor() as ex:
        res = [ex.submit(func, x) for x in args]

        for f in cf.as_completed(res):
                print(f.result())
```

MultiProcessing

Utilizzo

```
import  multiprocessing as mp

p   = mp.Process(target=func, args=[x, y])
p.start()
p.join()
```

```
import concurrent.futures as cf

with cf.ProcessPoolExecutor() as executor:
   secs = [5, 4, 3, 2, 1]
   results = executor.map(do_something, secs)

   for f in cf.as_completed(res):
        print(f.result())
```

JobLib

Feature

- Caching Trasparente di Funzioni (Memoizzazione)
- Calcolo Parallelo
- Debug Facile
- Multiparadigma

Utilizzo

```
from math import sqrt

from joblib import Parallel, delayed

Parallel(n_jobs=2)(delayed(sqrt)(i**2)
                   for i in range(10))
```

Honorable Mentions

Librerie

MRJob

Risorse

https://carpentries-incubator.github.io/lesson-parallel-python/aio/index.html

https://link-springer-com.proxy-ub.rug.nl/referenceworkentry/10.1007%2F978-0-387-09766-4_2

https://hpc.llnl.gov/training/tutorials/introduction-parallel-computing-tutorial

http://masnun.rocks/2016/10/06/async-python-the-different-forms-of-concurrency/

https://leimao.github.io/blog/Python-Concurrency-High-Level/

Calcolo Parallelo e Distribuito

Parte 2

Cezar Sas
c.a.sas@rug.nl

Comunicazione Intra-Processi

- Code/Pile
- Lock
- Condition

RECAP

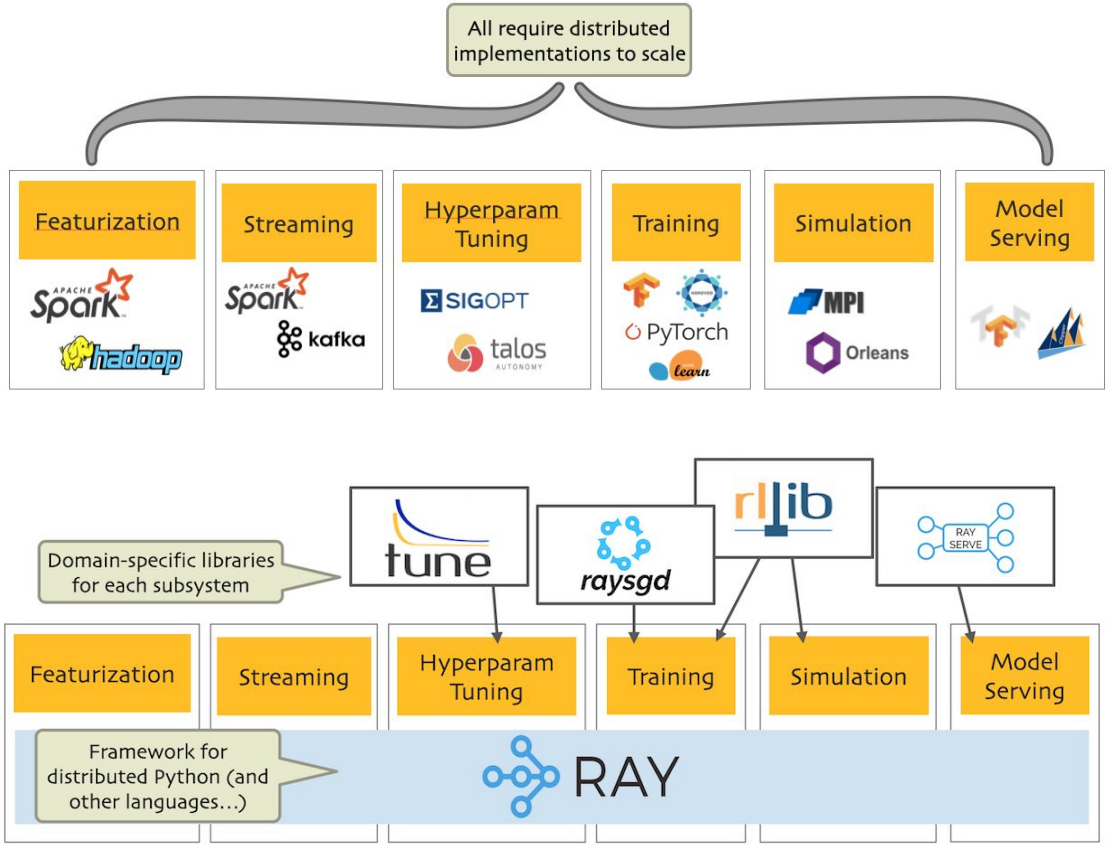
```
if io_bound:
   if io_very_slow:
      print("Use Asyncio")
   else:
     print("Use Threads")
else:
   print("Multi Processing")
```

Fonte: http://masnun.rocks/2016/10/06/async-python-the-different-forms-of-concurrency/

Ray

Fonte: AnyScale Academy

Perche'?


All require distributed implementations to scale
Featurization
Streaming
Hyperparam Tuning
Training
Simulation
Model Serving
Spark
hadoop
Spark
kafka
SIGOPT
talos
PyTorch
learn
MPI
Orleans
Domain-specific libraries for each subsystem
tune
raysgd
rllib
RAY SERVE
Framework for distributed Python (and other languages…)
RAY

API

- **ray.init()** # Inizializza l’applicazione ray
- **@ray.remote** # Trasforma le funzioni in Task, e le classi in Actor
- **x.remote()** # Crea un'istanza di un attore, o esegue un task o metodo di un attore
- **ray.put()** # “Sposta” un oggetto nello storage distribuito di oggetti
- **ray.get()** # “Prende” un oggetto dallo storage distribuito
- **ray.wait()** # Attende la fine dell’esecuzione di una list di task