Use ALL the cores!

Introduzione al multiprocessing in Python

Alessandro Di Federico

ale@clearmind.me

Politecnico Open unix Labs

18 giugno 2015

Indice

Cos’è il multithreading?

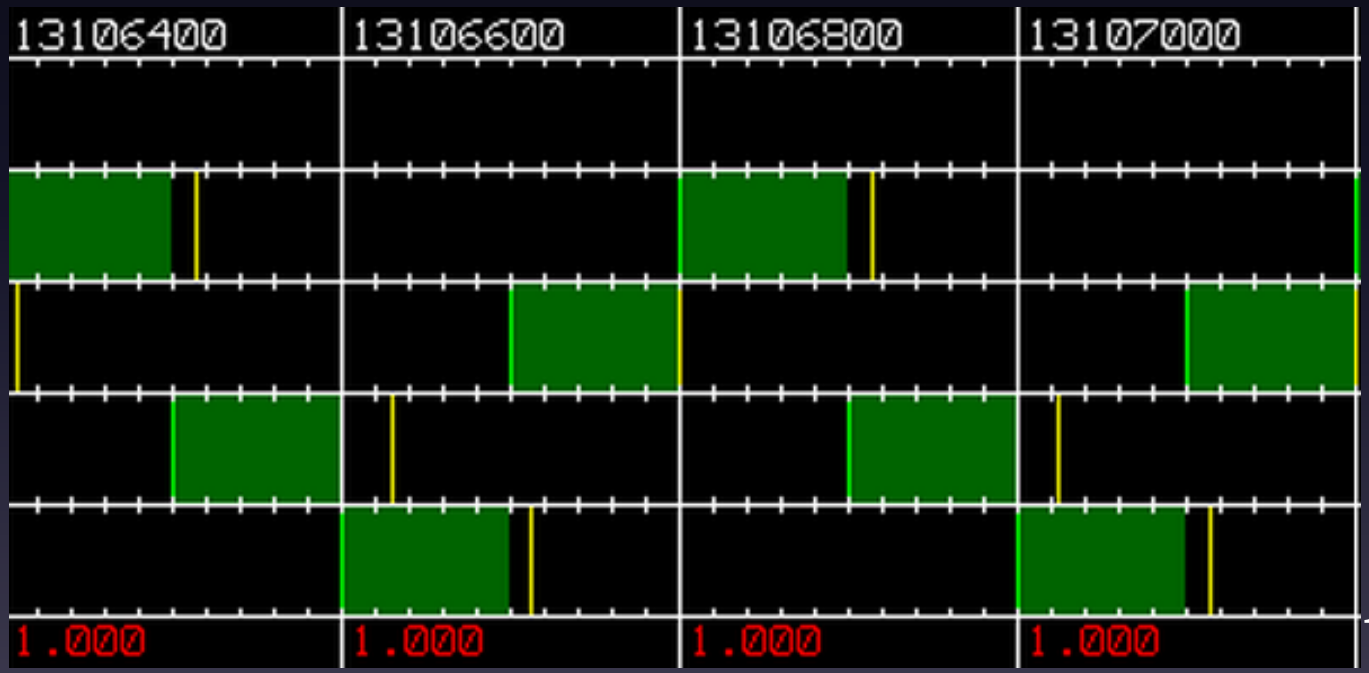
- Permette di compiere più operazioni in parallelo
- All’interno di una singola applicazione!
- Utile per sfruttare appieno le architetture multi-core
- Tutti i thread condividono la stessa memoria

La situazione in Python

- In Python il multithreading è supportato
- Tuttavia è sconsigliabile usarlo

La causa

Il Global Interpreter Lock



[1]

[1] http://www.dabeaz.com/GIL/gilvis/fourthread.html

Il GIL

- Non più di un thread può eseguire codice **Python**
- Il GIL viene rilasciato
  - in caso di I/O[2]
  - in caso di chiamata a librerie native (e.g. scritte in C)

[2] Il multithreading non andrebbe usato per attendere su I/O, un approccio asincrono è molto più efficace

`multiprocessing` to the rescue

- In alternativa si può usare il multiprocessing
- Il codice è eseguito in parallelo ma su processi distinti
- Istanze diverse di Python, GIL distinti!
- La memoria non è condivisa!

Indice

Da dove inizio?

- Il modulo `multiprocessing` contiene tutto il necessario
- Definiamo la funzione da eseguire in parallelo:

```
def hello(index):
    print("Hello from processor " + str(index))
```

Creiamo un processo

- La classe `Process` è quella che ci interessa:

```
from multiprocessing import Process

new_process = Process(target=hello, args=(1,))
new_process.start()
new_process.join()
```

- Il costruttore ha due parametri:
  - `target`: la funzione da eseguire sul nuovo processo
  - `args`: una tupla con gli argomenti da passare a `target`
- `start` avvia il processo
- `join` attende la sua terminazione

Un processo per core

```
from multiprocessing import Process, cpu_count

processes = []
for i in range(cpu_count()):
    new_process = Process(target=hello, args=(i,))
    processes.append(new_process)
    new_process.start()

for process in processes:
    process.join()
```

Risultato

```
$ python3 hello.py
Hello from processor 0
Hello from processor 1
Hello from processor 2
Hello from processor 3
Hello from processor 4
Hello from processor 5
Hello from prHello from processor 7
ocessor 6
```

Cos’è successo?

- Gli 8 processi sono stati eseguiti in parallelo
- L’esecuzione di un processo può essere interrotta
- E un altro potrebbe subentrare nel mezzo di un’operazione
- Il processo 6 è stato interrotto in favore del processo 7

I lock

- Un lock può essere *acquisito* e *rilasciato*
- Solo un processo alla volta può acquisirlo
- Gli altri si mettono in attesa

La classe `Lock`

```
from multiprocessing import Process, \
    cpu_count

processes = []
for i in range(cpu_count()):

    new_process = Process(target=hello, args=(i,))

    processes.append(new_process)
    new_process.start()

for process in processes:
    process.join()
```

La classe `Lock`

```
from multiprocessing import Process, \
    cpu_count, Lock

processes = []
for i in range(cpu_count()):
    output_lock = Lock()
    new_process = Process(target=hello, \
                          args=(i, output_lock))
    processes.append(new_process)
    new_process.start()

for process in processes:
    process.join()
```

Usiamo il lock

```
def hello(index):
    #
    print("Hello from processor " + str(index))
    #
```

Usiamo il lock

```
def hello(index, output_lock):
    output_lock.acquire()
    print("Hello from processor " + str(index))
    output_lock.release()
```

Riproviamo

```
$ python3 hello.py
Hello from processor 0
Hello from processor 1
Hello from processor 2
Hello from processor 3
Hello from processor 4
Hello from processor 5
Hello from processor 6
Hello from processor 7
```

Domande?

Indice

Cos'è?

- Preso un punto nel piano complesso c
- Detto $f_c(z) = z^2 + c$
- c è parte dell'insieme di Mandelbrot se

$$\sup_{n \in \mathbb{N}} |f_c^n(0)| < \infty$$

- In pratica:

$$f_c(f_c(f_c(f_c(f_c(f_c(f_c(f_c(0)))))))) < M$$

Visualizziamolo

- Definiamo un numero massimo di iterazioni n
- Calcoliamo al più n iterazioni della successione
- Ci fermiamo se:
  - superiamo il limite di iterazioni
  - il valore calcolato eccede il massimo M
- Il numero dell'iterazione $\hat{n}$ a cui usciamo determina il colore
- Ripetiamo per ogni punto del piano complesso

Implementazione seriale

```
from mandelbrot import set_size, compute_pixel
from PIL import Image

width, height = (560, 600)
set_size((width, height))

output = Image.new("RGB", (width, height))

for x in range(width):
    for y in range(height):
        pixel = compute_pixel((x, y))
        output.putpixel((x, y), pixel)

output.save("/tmp/out.png")
```

Quanto impiega?

```
$ python3 mandelbrot-sequential.py
It took 27.31 seconds
```

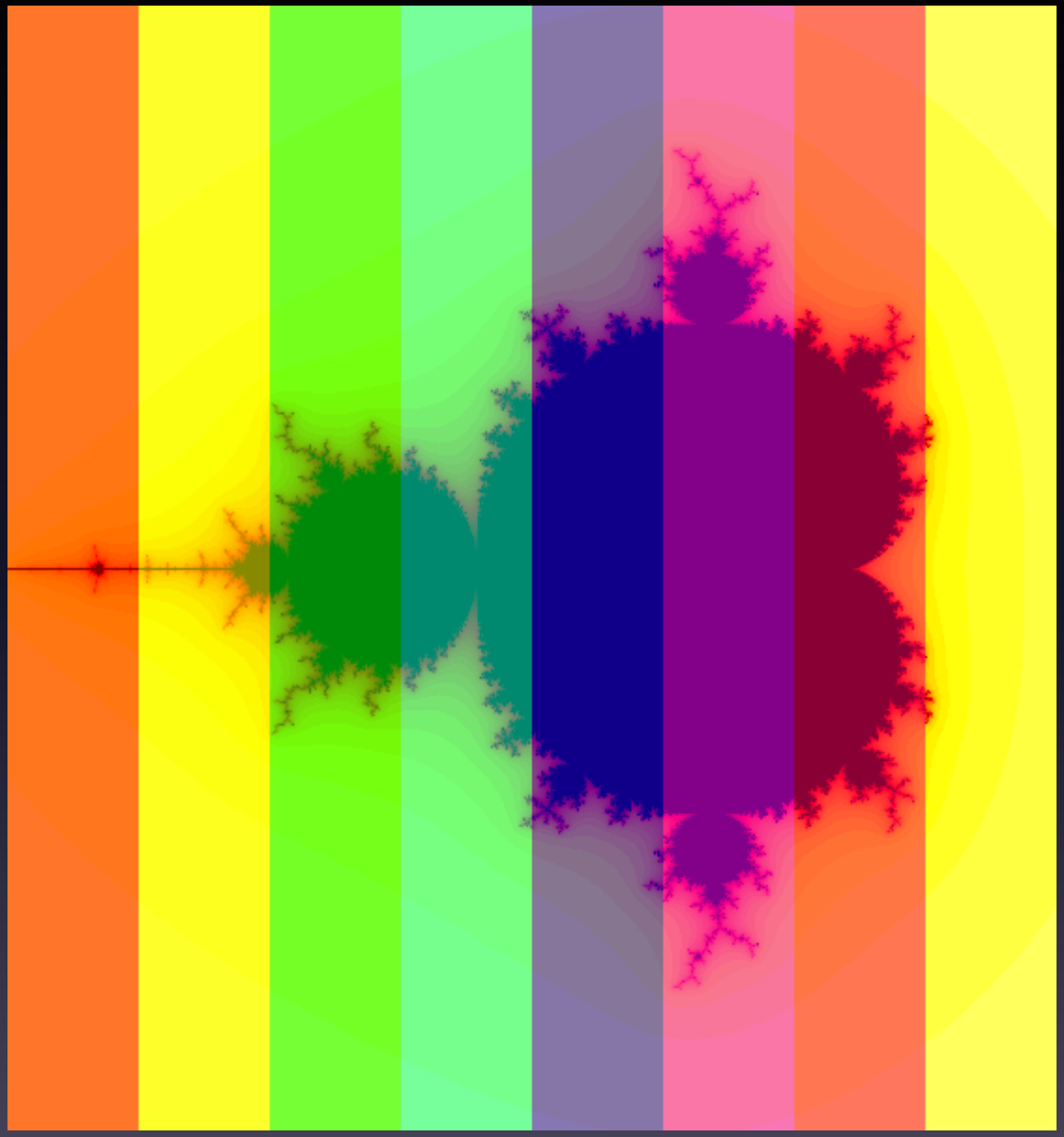
- Un po' lento, no?
- Ogni pixel può essere calcolato in maniera indipendente
- Possiamo parallelizzare!

Domande?

Indice

Un primo approccio

- Possiamo dividere l'immagine in fette
- Ogni processo calcolerà i pixel di una fetta
- Il processo master alla fine raccoglierà i risultati



Le pipe

- Per comunicare con i processi useremo le `Pipe`
- Un canale di comunicazione tra due processi:
  - `send`: invia un oggetto tramite la pipe
  - `recv`: riceve un oggetto inviato da una `send`

Il processo master (parte 1)

```
from multiprocessing import Pipe, Process, \
    cpu_count

cpu_count = cpu_count()
width, height = (560, 600)
slice_width = int(width / cpu_count)

processes = []
for i in range(cpu_count):
    parent_pipe, child_pipe = Pipe()
    new_proc = Process(target=compute_slice, \
        args=(i, slice_width, child_pipe))

    process_info = (i, new_proc, parent_pipe)
    processes.append(process_info)
    new_process.start()
```

I processi figli

```
def compute_slice(index, slice_width, pipe):
    result = []
    for y in range(height):
        start_x = index * slice_width
        end_x = start_x + slice_width
        for x in range(start_x, end_x):
            pixel = compute_pixel((x, y))
            result.append(pixel)

    pipe.send(result)
    pipe.close()
```

Il processo master (parte 2)

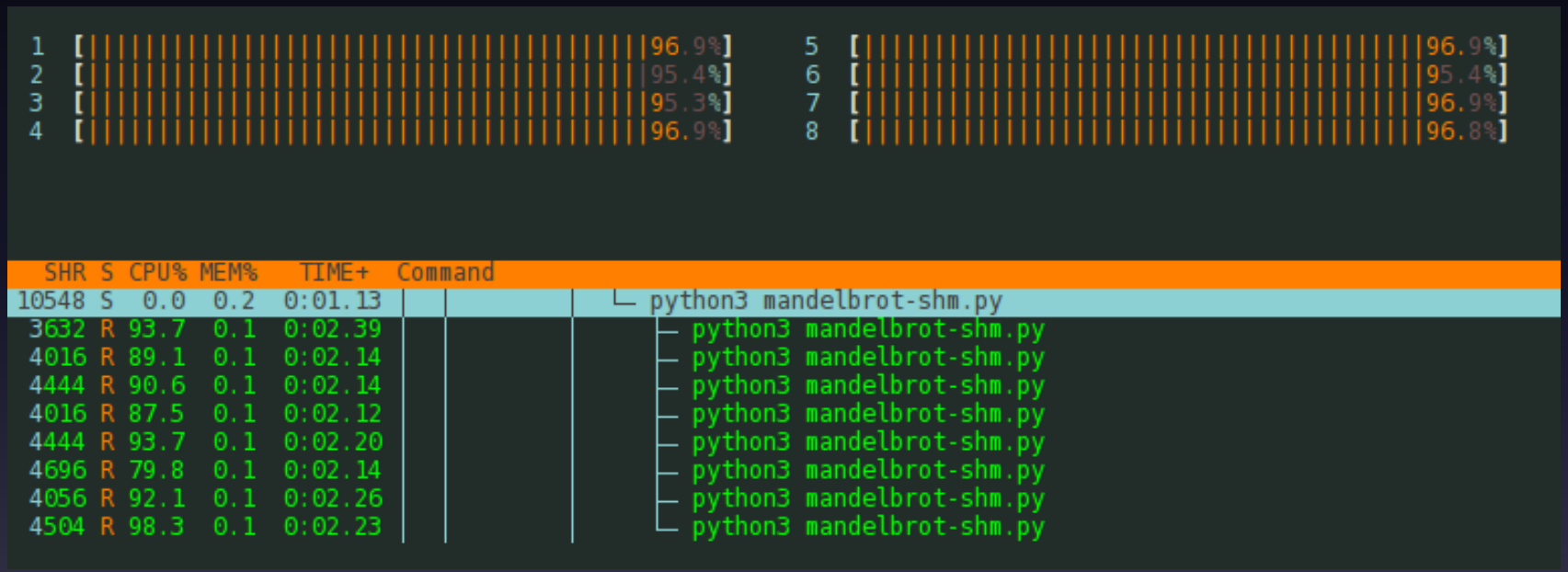
```
for process_index, process, pipe in processes:
    result = pipe.recv()
    process.join()

    for pixel_index, pixel in enumerate(result):
        x = int(pixel_index % slice_width) \
            + process_index * slice_width
        y = int(pixel_index / slice_width)
        output.putpixel((x, y), pixel)
```

Proviamolo

```
$ python3 mandelbrot-split-image.py
Process 0 took 1.144 seconds
Process 1 took 1.291 seconds
Process 2 took 2.561 seconds
Process 3 took 2.719 seconds
Process 4 took 4.088 seconds
Process 5 took 5.331 seconds
Process 6 took 3.357 seconds
Process 7 took 1.118 seconds
It took 6.453 seconds
```

Finalmente

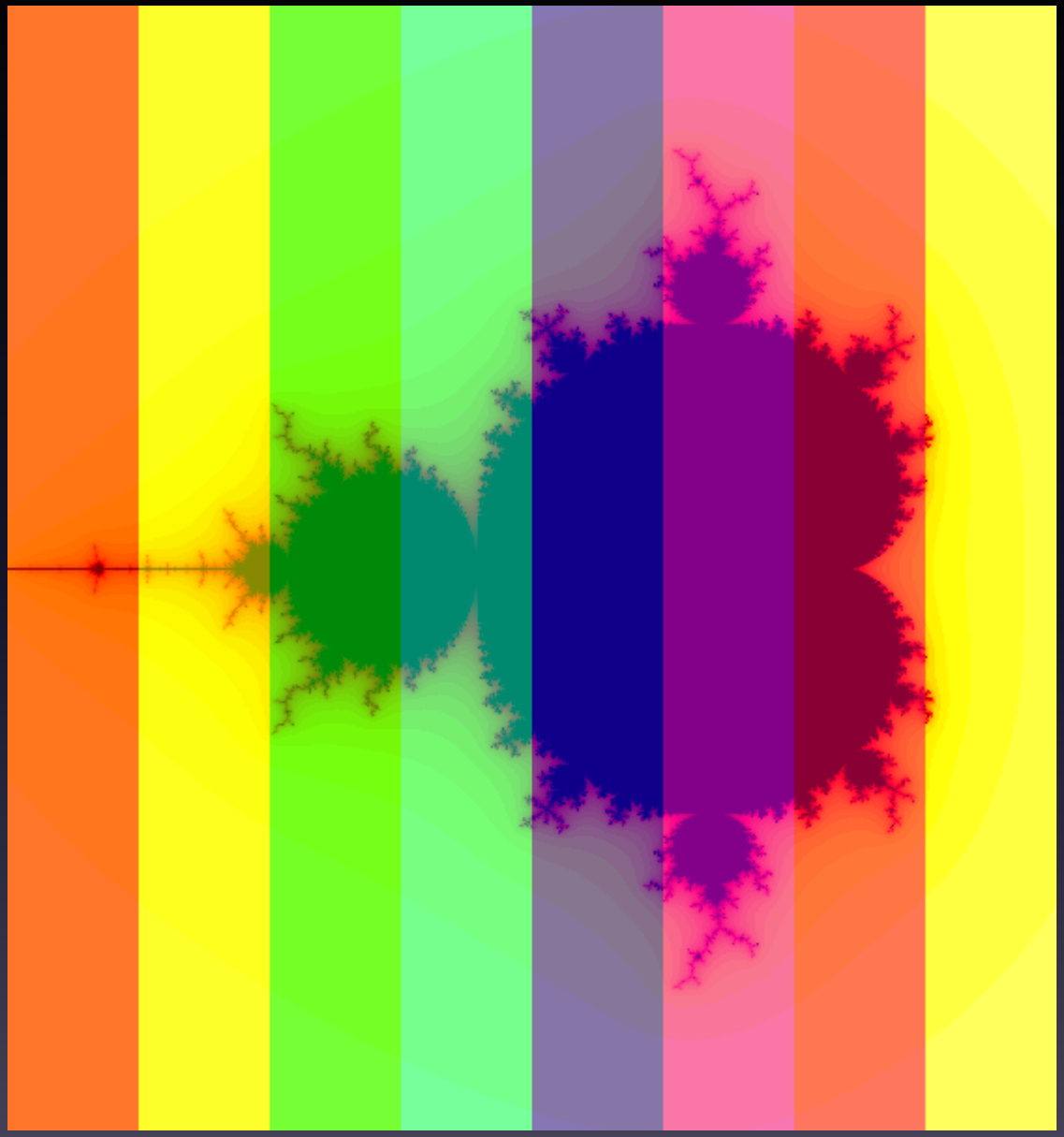

1 [||||||||||||||||||||||||||||||||||||||||96.9%]
2 [||||||||||||||||||||||||||||||||||||||||95.4%]
3 [||||||||||||||||||||||||||||||||||||||||95.3%]
4 [||||||||||||||||||||||||||||||||||||||||96.9%]
5 [||||||||||||||||||||||||||||||||||||||||96.9%]
6 [||||||||||||||||||||||||||||||||||||||||95.4%]
7 [||||||||||||||||||||||||||||||||||||||||96.9%]
8 [||||||||||||||||||||||||||||||||||||||||96.8%]
SHR S CPU% MEM% TIME+ Command
10548 S 0.0 0.2 0:01.13 └─ python3 mandelbrot-shm.py
3632 R 93.7 0.1 0:02.39 ├─ python3 mandelbrot-shm.py
4016 R 89.1 0.1 0:02.14 ├─ python3 mandelbrot-shm.py
4444 R 90.6 0.1 0:02.14 ├─ python3 mandelbrot-shm.py
4016 R 87.5 0.1 0:02.12 ├─ python3 mandelbrot-shm.py
4444 R 93.7 0.1 0:02.20 ├─ python3 mandelbrot-shm.py
4696 R 79.8 0.1 0:02.14 ├─ python3 mandelbrot-shm.py
4056 R 92.1 0.1 0:02.26 ├─ python3 mandelbrot-shm.py
4504 R 98.3 0.1 0:02.23 └─ python3 mandelbrot-shm.py

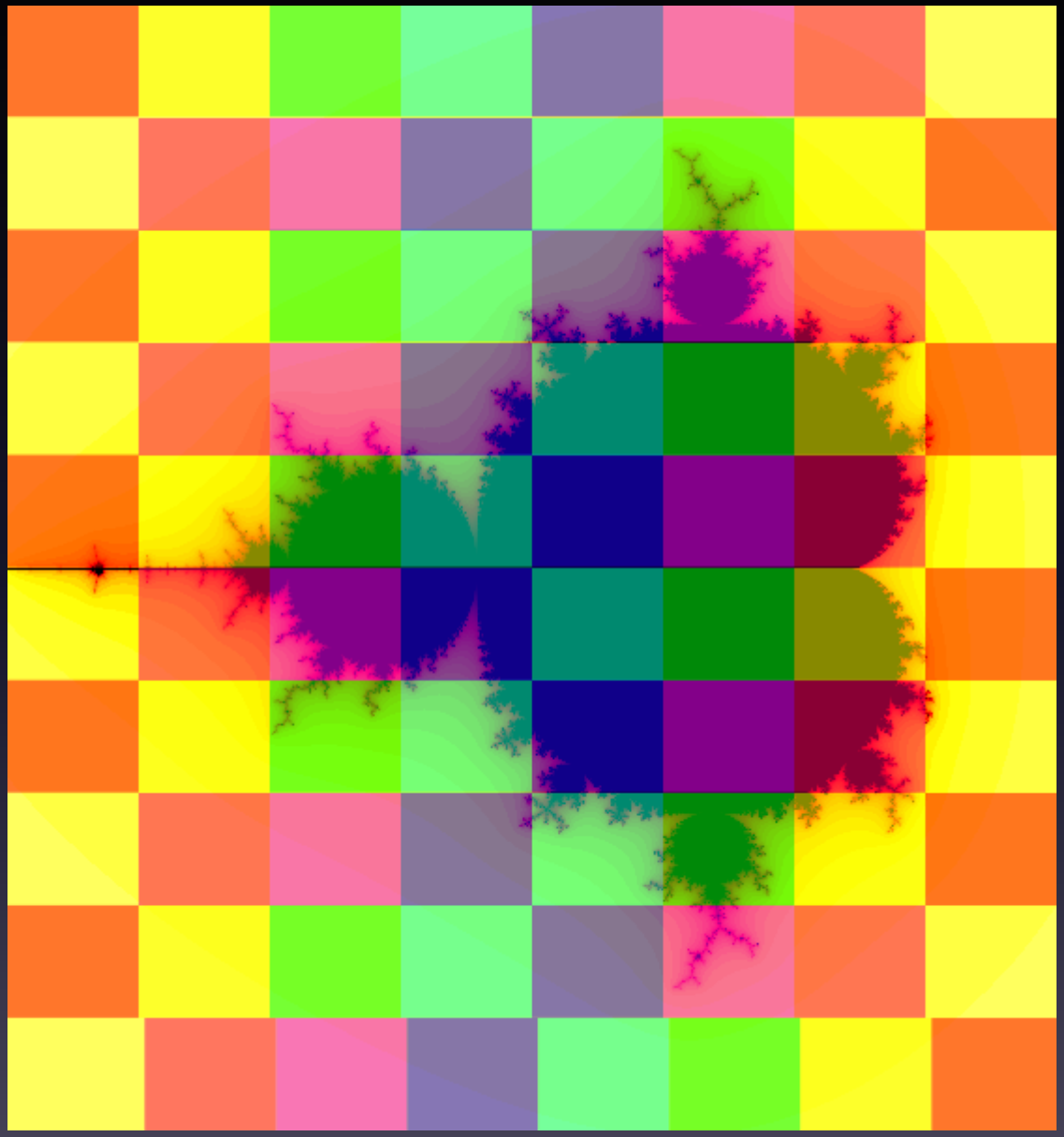
Domande?

Indice

Unità di lavoro più piccole

- Alcune zone del Mandelbrot sono più complesse di altre
- Le tecnica «a fette» distribuisce male il carico di lavoro
- Un'idea è lavorare su piccole porzioni dell'immagine
- Finita una, si va alla successiva





Il pool

- La classe `Pool` offre un «bacino» di *worker* process
- Essi sono a disposizione per fare computazione

La funzione `map`

- Due parametri: una lista *L* e una funzione *f*
- Applica *f* a ciascun elemento di *f*
- Restituisce una lista con i nuovi valori

```
def aggiungi1(x):
  return x + 1

map(aggiungi1, [3,4,5])
# Risultato: [4, 5, 6]
```

`Pool.map`

- Pool offre una versione *parallela* di `map`
- Ogni worker libero calcola un elemento
- Possiamo usarlo per calcolare i nostri blocchi

Creiamo il pool

```
from multiprocessing import Pool, cpu_count

width, height = (560, 600)
cpu_count = cpu_count()
block_width, block_height = (70, 60)

pool = Pool(processes=cpu_count)

total_blocks = int((width * height) / \
    (block_width * block_height))

pixel_blocks = pool.map(compute_block, \
    range(total_blocks))
```

La funzione `compute_block`

```
def compute_block(block_index):
    result = []

    start_x = # ...
    end_x = start_x + block_width

    start_y = # ...
    end_y = start_y + block_height

    for x in range(start_x, end_x):
        for y in range(start_y, end_y):
            pixel = compute_pixel((x, y))
            result.append(pixel)

    return result
```

Proviamo

```
$ python3 mandelbrot-pool.py
It took 4.294 seconds
Process 0 took 3.223 seconds  (9 blocks)
Process 1 took 3.232 seconds  (6 blocks)
Process 2 took 3.544 seconds (12 blocks)
Process 3 took 4.250 seconds  (6 blocks)
Process 4 took 3.621 seconds  (6 blocks)
Process 5 took 3.009 seconds (17 blocks)
Process 6 took 2.947 seconds  (9 blocks)
Process 7 took 3.010 seconds (15 blocks)
```

Domande?

Indice

Un passo ulteriore

- Finora i risultati sono passati tramite una pipe
- Questo passaggio dati tra processi ha un costo
- Python permette di condividere un’area di memoria

La classe `Array`

- Rappresenta memoria condivisa
- Prende un tipo di dati e uno stato inziale
- Per un singolo valore si veda `Value`
- Può essere passata come parametro tra processi

```
Array("i", [1,2,3,4])
```

Inizializziamo la memoria condivisa

```
image_data = None

def init_process(shared_array):
    global image_data
    image_data = shared_array

size = width * height
shared_array = Array("i", [0] * size * 3)
pool = Pool(processes=cpu_count, \
    initializer=init_process, \
    initargs=(shared_array,))

# Pool.apply è come map, ma non da risultato
pool.apply(compute_block, range(total_blocks))
```

La nuova `compute_block`

```
def compute_block(block_index):
    start_x = # ...
    end_x = start_x + block_width
    start_y = # ...
    end_y = start_y + block_height
    for x in range(start_x, end_x):
        for y in range(start_y, end_y):
            pixel = compute_pixel((x, y))
            index = 3 * (x + y * width)
            image_data[index + 0] = pixel[0]
            image_data[index + 1] = pixel[1]
            image_data[index + 2] = pixel[2]
```

Performance?

```
$ python3 mandelbrot-shm.py
It took 4.741 seconds
Process 0 took 4.329 seconds  (9 blocks)
Process 1 took 4.447 seconds (15 blocks)
Process 2 took 4.459 seconds (12 blocks)
Process 3 took 4.442 seconds  (8 blocks)
Process 4 took 4.604 seconds  (9 blocks)
Process 5 took 4.737 seconds  (6 blocks)
Process 6 took 4.572 seconds  (6 blocks)
Process 7 took 4.296 seconds (15 blocks)
```

Qualcosa è andato storto

Qualcosa è andato storto

Ogni accesso ad un elemento di `Array` è locked

Rimuoviamo il lock

```
size = width * height
shared_array = Array("i", [0] * size * 3, \
    lock=False)
```

Molto meglio

```
$ python3 mandelbrot-shm.py
It took 3.601 seconds
Process 0 took 3.181 seconds (12 blocks)
Process 1 took 3.067 seconds (12 blocks)
Process 2 took 3.169 seconds  (9 blocks)
Process 3 took 3.063 seconds  (6 blocks)
Process 4 took 3.596 seconds  (6 blocks)
Process 5 took 3.019 seconds (23 blocks)
Process 6 took 3.455 seconds  (6 blocks)
Process 7 took 3.405 seconds  (6 blocks)
```

Solo un pezzo della storia

- Abbiamo visto solo esempi *embarassing parallel*
- IRL ci possono essere dipendenze dati
- Si possono creare data race e quindi incoerenze
- L'uso dei lock va studiato con attenzione

Domande?

Licenza