Pendolo quadrifilare e multithreading

Francesco Sermi

10 giugno 2024

Indice

Capitolo 1

Introduzione

Il senso di questa breve (o almeno spero lo sia, ma ai posteri viene lasciata l'ardua sentenza, ovvero al "Francesco" del futuro che avrà terminato questo documento) introduzione nasce dal rispondere alla domanda consequenzialmente naturale nell'esatto istante in cui si legge il titolo di questo documento: *perché* scrivere un documento del genere?
La prima ragione è dovuta al fatto che mi piace raccogliere la *conoscenza* (qualunque essa sia) nella forma più elegante possibile, in maniera tale da essere facilmente reperibile non solo da me, ma anche *visivamente* nel caso in cui mi servisse più avanti in futuro.
La seconda ragione deriva dal fatto che, scrivendo il programma, mi sono reso conto che dei piccoli periodi scritti con la funzionalità dei commenti non mi consentiva di spiegare al meglio quello che avevo fatto, al contrario di un bellissimo documento LaTeX, dove ho la possibilità di mettere il codice affianco e quella di usufruire di qualunque tipo di risorsa grafica per esprimere un concetto.
La sfida che mi sono posto era quella di scrivere una possibile implementazione del fit ai minimi quadrati utilizzando il linguaggio di programmazione Python, ma utilizzando un approccio multi-threading per ottimizzare ancora di più il tutto.

1.1 Il multi-threading

Python è uno fra i linguaggi di programmazione più popolari attualmente: il motivo di questa grandissima fama di cui gode deriva da una serie di ragioni

① **è linguaggio di programmazione ad alto livello**: Python è uno dei linguaggi che più si avvicina al nostro modo di pensare, dal punto di vista paradigmico, dunque è facilmente apprezzabile dai neofiti della programmazione piuttosto che un C++ dove bisogna la mancanza di un *garbage collector* (automatico almeno, siccome nello standard GCC del 2017 è stato introdotto, nella libreria standard, una versione rudimentale che va però invocata) e la sintassi non proprio ovvia lo rende complicato da studiare (almeno per le persone alle prime armi);

② **sintassi simile allo pseudocodice**: dal punto di vista della sintassi, Python è imbattile, siccome sempre di star leggendo una ricetta

③ **Facilità di installazione**: Python è facilmente reperibile su qualunque sistema operativo in maniera nativa, senza dover appoggiarsi a dei *porting* dei compilatori (come nel caso del C++);

④ **Librerie**: Python, a causa della sua larghissima diffusione, gode di una collezione ricchissima di librerie, che spaziano dal Machine Learning, Deep Learning all'analisi statistica.

Proprio su quest'ultimo punto mi voglio soffermare un attimo, siccome è strettamente connesso al concetti di multithreading, necessario per la programmazione parallela di cui noi abbiamo bisogno. L'implementazione di riferimento[1] di Python è il **CPython** la cui caratteristica fondamentale è il fatto che fa uso del cosiddetto **GIL**:

[1] ovvero lo standard che viene considerato come riferimento in base a cui tutte le altre implementazioni dello stesso standard sono valutate e rispetto alla quale tutti i miglioramenti sono aggiunti

Definizione 1.1: Global Interpreter Lock

Il **GIL** è un meccanismo usato nei linguaggi di programmazioni interpretati (e non compilati) per sincronizzare l'esecuzione dei thread così che un solo thread (quello nativo che si occupa dell'esecuzione del processo programma) possa eseguire operazioni base (quelle dei calcolatori s'intende, quindi allocazione della memoria e, in generale, il *reference counting*)

Sebbene possa sembrare limitante ad una prima analisi, di per sé, questo meccanismo garantisce una serie di vantaggi non da poco:

- il GIL va a incrementare di molto la velocità di un programma a singolo thread, siccome il computer non si deve occupare di gestire l'accesso dei vari thread che accedono alle stesse *data structures*;
- garantisce una maggiore facilità di implementazione della librerie C che contengono codice non *thread-safe*;
- è facile da implementare, infatti è molto più semplice implementare un interprete con GIL piuttosto che un interprete con funzionalità *multi-threading*

La domanda adesso nasce spontanea: *perché accingersi ad implementare il multi-threading?* Banalmente per usare al meglio le risorse del computer: sebbene per programmi *leggeri* l'utilizzo del multi-threading spesso va a sfavorire le performance piuttosto che aumentarle (almeno in Python) per applicazioni in cui bisogna effettuare delle operazione che sono computazionalmente pesanti offre dei vantaggi non indifferenti.
La parte teorica spero adesso che sia finita.

1.2 Multithreading e statistica: un buon connubio

Già solo effettuando il corso di Laboratorio I (ma in generale anche per i miei hobby, come quando ho tentato di generare su Python dei frattali autosimili con `plotter` e vettori da 10000 elementi oppure quando ho partecipato al corso avanzato di Python e abbiamo simulato l'equazione di Schrodinger) mi sono sempre più accorto di quanto la realizzazione di alcuni di questi programmi facessero un larghissimo utilizzo di cicli annidati, che sono un grande problema: infatti, finché si parla di pochi dati non è in sé un problema, ma nel momento in cui sui singoli dati di un array bisogna effettuare numerosi cicli e accessi ad altre celle di memoria si vedono che le performance di Python non sono nemmeno paragonabili a quelle dei linguaggi a basso livello (sebbene, ahimé, sacrifichino quella facilità di lettura).
In statistica si ha sempre a che fare con delle strutture di dati con grandi dimensioni: non a caso, negli ultimi anni, è nata la figura del *big data scientist*[2]. Dunque, mi sono posto la sfida, come avevo accennato, di implementare il fit ai minimi quadratici utilizzando la programmazione parallela. Solitamente nei fit si utilizzano 3 ipotesi di lavoro:

① le misure sono fra loro indipendenti;

② i valori y_i che vogliamo fittare sono calcolati in corrispondenza di x_i noti, ovvero le incertezze di misura che abbiamo sulle x_i sono trascurabili rispetto a quelle sulle y_i:

$$\left|\frac{df}{dx}(x_i;\hat{\theta}_1,\dots,\hat{\theta}_m)\sigma_{x_i}\right| \ll \sigma_{y_i}\ \forall i = 1,\dots,n \tag{1.1}$$

③ gli errori di misura sulla variabile dipendente (ovvero gli y_i) sono gaussiani, ovvero tendono a fluttuare attorno al loro valor medio con una distribuzione gaussiana con deviazione standard σ_{y_i} nota a priori

Sebbene per quanto riguarda la teoria, tutte queste sono affermazioni molto forti e molto importanti (siccome garantiscono che il χ^2 che definiremo sotto sia distribuito come una gaussiana a sua volta, eccetera); nella pratica[3] ce n'è una a cui si deve prestare particolare attenzione, ovvero la seconda ipotesi: in tal caso infatti il metodo dei

[2] la figura che, dal punto di vista aziendale, si occupa di analizzare campioni di dati di grandi dimensioni

[3] per tutti coloro che leggeranno questo documento oppure al me del futuro: l'intento di questa frase non è provocatoria, perché di fatto sono tutte egualmente importanti tuttavia se le prime due condizioni non sono rispettate le stime dei parametri effettuate dal fit potrebbero comunque essere ragionevoli (ma le incertezze su di essi potrebbero non esserlo), mentre la seconda ipotesi può spesso condurre a delle stime completamente sbagliate dei parametri

minimi quadrati utilizzato solitamente da librerie come `scipy` è sbagliato, siccome tiene di conto esclusivamente della distanza verticale fra le nostre misure e il modello, trascurando completamente quella orizzontale. In tal caso si utilizzano altri modelli di fit, come l'ODR (*Orthogonal Distance Regression*), oppure si può tentare di recuperare, in parte, il metodo dei fit ai minimi quadrati della libreria `curve_fit` di `scipy.optmize` utilizzando il metodo degli errori efficaci, in cui si propagano gli errori dei x_i moltiplicandoli per la derivata del modello utilizzato per il fit sommandoli in quadratura con con gli errori sulla y_i: in questo caso, facciamo una prima stima dei parametri che facciamo stimare dal modello con gli errori non efficaci e, successivamente, iteriamo il metodo (solitamente converge abbastanza velocemente, dunque bastano pochi passi per ottenere una stima già migliore dei parametri).
Tenterò, per quanto possibile, di ottenere un'implementazione che sia il più generale possibile, però attaccherò il problema passo per passo, cercando di generalizzarlo, per quanto possibile, al problema più generale di fit.
Iniziamo ad addentrarci: il principio più basilare su cui si basa l'idea di fit è quella di minimizzare la seguente quantità

$$\chi^2 = \sum_{k=1}^{n} \left(\frac{y_i - f(x_i; \hat{\theta}_1, \dots, \hat{\theta}_m)}{\sigma_{y_i}} \right)^2 \tag{1.2}$$

che, come si può intuire, rappresenta la distanza fra il modello e i dati della popolazione che stiamo studiando, pesata con le opportune incertezze.
Di fatto, quello che facciamo ogni volta che fittiamo dei dati, è quello di trovare un minimo ad una funzione dipendente da m parametri. Ora, finché sono fit lineari (ovvero la relazione che lega i parametri è lineare), possiamo trovare una soluzione in forma chiusa al nostro problema, altrimenti bisogna utilizzare dei metodi numerici (come, per esempio, le simulazioni Monte-Carlo).
Se vogliamo quindi implementare un metodo per implementare il fit, dobbiamo creare una griglia (o meglio uno spazio $E \subset \mathbb{R}^m$ nel caso di m parametri) in cui limitiamo i nostri parametri e data una m-upla di essi ci calcoliamo il χ^2

Capitolo 2

Programmazione parallela

Il concetto di programmazione parallela, che già dal nome si può facilmente intuire, può essere definito come

Definizione 2.1: Programmazione parallela

Con **programmazione parallela** si intende un tipo di computazione in cui molti calcoli o, in generale, molti processi sono eseguiti in maniera simultanea

Ovviamente il nostro approccio sarà rivolto, per lo più, all'implementazione di essa in Python, ma in generale questo problema ha una natura che si può tranquillamente astrarre da quella del linguaggio di programmazione che stiamo utilizzando[1].
Noi cerchiamo di implementarla in Python, dove la situazione non è proprio *out-of-the-box* a causa della presenza del **GIL** che in Python non può essere disattivato in alcuna maniera[2]. L'unico modo per implementare parzialmente in Python la programmazione parallela è la programmazione multithread, che ho deciso di utilizzare al posto della programmazione multiprocessuale.
Andiamo con ordine, cercando di spiegare il problema nella maniera più generale possibile. I programmi paralleli possono essere progettati in 2 maniere possibili:

- **Memoria condivisa**: nella memoria condivisa, le sotto-unità posso comunicare fra di loro tramite lo stesso spazio di memoria. Il vantaggio di questo approccio è il fatto che non deve essere gestita la comunicazione in maniera esplicita, siccome in questa maniera possiamo andare a leggere e scrivere senza problemi nello spazio condiviso. Il problema inizia a formarsi nel momento in cui più sub-unità cercano di accedere e cambiare la stessa porzione di memoria nello stesso momento e, per ovviare a ciò, si devono utilizzare delle tecniche di sincronizzazione.
- **Memoria distribuita**: nella memoria distribuita, ogni processo opera sul suo personale spazio di memoria. In questo caso, la comunicazione è gestita in maniera esplicita fra i processi e dal punto di vista computazione è dunque più costosa.

A causa del GIL tutte le librerie presenti riguardo al threading, purtroppo, sono di fatto inutili, siccome l'interprete è comunque progettato affinché non venga eseguito più di un thread parallelamente, sebbene non impedisca che non si possano creare (sebbene non garantiscano dei grandi vantaggi). Per questo utilizzeremo la seconda architettura (ed eventualmente rimanderemo un'implementazione multi-threading col C++), sebbene computazionalmente più costosa rispetto al threading.
La libreria che consente in Python di creare più processi è `multiprocessing`, di cui illustrerò le principali caratteristiche, funzionalità qua sotto; per poi dare una breve panoramica a come si utilizza. La libreria propone un'interfaccia simile a quella di un API che si basa sul creare un'istanza dell'oggetto `Process`, in cui bisogna specificare la funzione che dovrà andare ad eseguire. A seconda dalla piattaforma utilizzata, la libreria supporta tre modi per iniziare un processo:

- *spawn*: il processo "*padre*" crea un processo dell'interprete di Python e il processo figlio erediterà tutte quelle risorse che sono necessarie per eseguire il processo `run()` che vedremo in seguito.

[1] si potrebbe persino tentare di fare su linguaggi di programmazione esoterici come il Brainfuck
[2] sebbene nelle librerie di Python scritte C/C++ si può disattivare manualmente all'interno, tuttavia quando il controllo viene riassunto dalla Python VM dev'essere nuovamente sottoposta al controllo del GIL per non mandare un errore

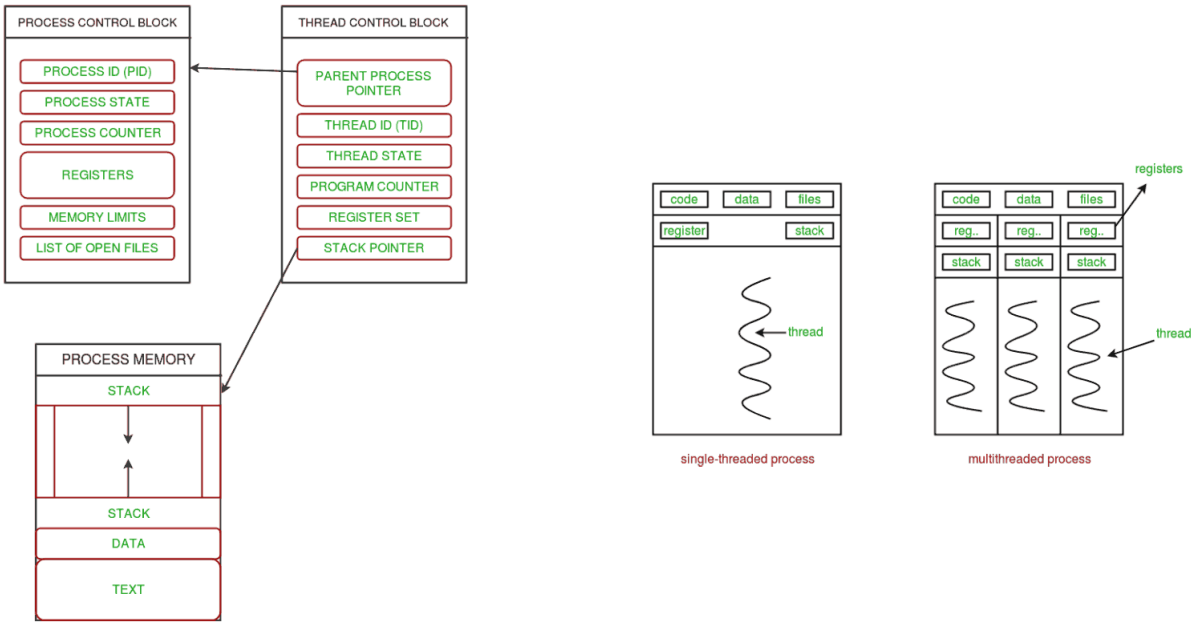

Figura 2.1: Funzionamento di un thread: esso contiene tutte le informazioni dentro il TCB(**Thread Control Block**) è identificato da un id che è unico, uno *stack pointer* che punta invece alle variabili che sono definite dentro il namespace del thread, un parametro che identifica lo stato del thread e il *parent process pointer* che invece punta al

- *fork*: il processo "figlio" andrà ad effettuare un `os.fork()`[3] all'interprete di Python, che quando inizierà sarà perfettamente identico a quello padre (infatti ne erediterà tutte le risorse) sebbene sia problematico effettuare un forking in piena sicurezza
- *forkserver*: quando il programma viene avviato ed è stato selezionato questa modalità, il processo server viene creato. Da quel punto in poi, quando un nuovo processo viene eseguito, il processo parente si connette al server e richiede che generi un processo nuovo, che sarà *single-threaded* a meno che non intervengano delle librerie di sistema che li andranno a generare.

```
1  import multiprocessing as mp
2
3  def hello(q):
4    q.put("Hello")
5  if __name__ == "__main__":
6    ctx = mp.getcontext("spawn")
7    q = ctx.Queque()
8    p = ctx.Process(target=foo, args=(q,))
9    p.start()
10   print(q.get())
11   p.join()
12
```

Figura 2.2: Esempio basilare di *spawn* di un processo: un "processo" è semplicemente un'istanza della classe Oggetto, da cui è possibile andare a definire che argomenti passargli o altro

Come si può intuire da questo listato qua accanto, i processi sono creati utilizzando un oggetto `Process` e chiamando il metodo `.start` da cui possiamo, per esempio, andare a passare alcuni argomenti necessari alla funzione.

[3]questa funzione si occupa di andare a generare un processo figlio perfettamente identico a quello al processo da cui è stata chiamata questa funzione