Programmazione Avanzata e parallela

Lezione 30

Luca Manzoni

Python e parallelismo

GIL, multiprocessing, Joblib

- I problemi di python nel multithreading
- Parallelismo con multiprocessing
  - Gestire i processi singolarmente
  - Pool
- Parallelismo usando joblib

Global Interpreter Lock

GIL

- Il GIL è un lock che protegge l'accesso a tutti gli oggetti Python
- Quindi un solo thread può interpretare il bytecode Python (il formato intermedio in cui viene "compilato" Python)
- Le operazioni che non sono l'interpretazione del codice Python possono avvenire in parallelo (I/O, utilizzo di operazioni numpy, etc)…
- …ma l'esecuzione di codice Python puro è comunque limitata dal fatto che un solo thread alla volta può eseguirla

Global Interpreter Lock

Soluzioni

- Quando usiamo librerie non legate al GIL (e.g., che chiamano codice C) non vi sono restrizioni sul parallelismo (e.g., possiamo chiamare una libreria che usa OpenMP)

- Quando abbiamo molte operazioni di I/O che dominano il tempo di esecuzione l'effetto del GIL è minore

- Se eseguiamo codice Python puro in più **processi** possiamo eseguire in parallelo:

  - Ogni processo ha un suo interprete con un suo GIL

Parallelismo in Python

Librerie

- È possibile utilizzare delle librerie che permettono di lavorare con più processi:
  - Incluso in Python è il modulo “multiprocessing”
  - Come libreria esterna esiste “joblib” (https://joblib.readthedocs.io/en/stable/)

Parallelismo in Python

Multiprocessing

- La librerie multiprocessing permette di creare e controllare processi:
    - Oggetti **Process** che possono essere avviati
    - Oggetti **Pool**, rappresentanti un insieme di processi a cui è possibile delegare delle operazioni da svolgere

Parallelismo in Python

Multiprocessing

- Process:
  - Creato con **Process(target=funzione, args=(….))**
    - Il target è la funzione da eseguire
    - Args è una tupla di argomenti
    - Una volta creato non è ancora attivo, abbiamo solo detto cosa dovrà eseguire
- Avviato col metodo **.start()**
- Possiamo attendere che termini con **.join()**

Parallelismo in Python

Multiprocessing

- Queue:
  - Creata con **Queue()**
  - Rappresenta una coda utilizzabile da più processi per inviare e ricevere dati:
    - **q.put(oggetto)** per aggiungere alla coda (possibilmente bloccante)
    - **q.get()** per ottenere dalla coda (possibilmente bloccante)
    - È possibile scegliere di avere una eccezione invece di una operazione bloccante

Parallelismo in Python

Multiprocessing

- Lock:
  - Creato con **Lock()**
  - Rappresenta un mutex che è possibile acquisire con **.acquire()**
  - e rilasciare con **.release()**
  - Come per ogni lock vi è il rischio di deadlock!
  - Chiaramente diventa più utile in caso di risorse condivise

Parallelismo in Python

Multiprocessing

- Value e Array:
  - Value e Array rappresentano un modo di condividere memoria tra più processi:
    - **Value(typecode, valore_iniziale)**
      dove typecode può essere ‘i’ (intero), ‘d’ (floating point a precisione doppia), ‘f’ (floating point a precisione singola)
    - **Array(typecode, valore_iniziale)**
      dove il valore iniziale sono i valori dell’array
- In entrambi i casi di default ognuno di questi ha un lock che protegge l’accesso alla variabile

Parallelismo in Python

Multiprocessing

- Pool:
  - Rappresenta un insieme di processi a cui possiamo delegare di svolgere dei compiti
  - Creato con **Pool(processes=num_processes)**
  - Il metodo **.map(funzione, iterabile)** distribuisce l’applicazione della funzione a tutti gli elementi tra i diversi processi
  - È possibile usare **.close()** per liberare le risorse una volta che tutti i processi hanno terminato i task assegnati

Parallelismo in Python

Joblib

- Per parallelizzazione di cicli joblib prevede una interfaccia comoda e più metodi di parallelizzazione:

- **Parallel(n_jobs=num_jobs)(iterabile)**

- In questo caso la funzione da applicare a ogni elemento è bene sia racchiusa in “delayed”:

  - Invece di **f(x)**…

  - **delayed(f)(x)**

  - Questo indica che vogliamo applicare f ma non applicarla “ora”

Just-In-Time Compilation

Numba

- La compilazione just-in-time (JIT) è una compilazione compiuta durante l'esecuzione del programma
- Si spende del tempo a compilare…
- …ma se il codice è eseguito più volte questo potrebbe velocizzare l'esecuzione
- Una librerie che fornisce questa funzionalità a Python è **numba** (che usa LLVM)
- Tramite un decoratore è possibile compilare una funzione in codice nativo: **@jit**