Programmazione Avanzata

Concorrenza: I parte

11

Tipi di concorrenza e dati condivisi

- I diversi modi di implementare la concorrenza si differenziano principalmente per il modo in cui vengono condivisi i dati:
  - accesso diretto ai dati condivisi, ad esempio attraverso memoria condivisa
  - accesso indiretto, ad esempio, usando la comunicazione tra processi (IPC)
- La concorrenza a thread consiste nell'avere thread concorrenti separati che operano all'interno di uno stesso processo. Questi thread tipicamente accedono i dati condivisi attraverso un accesso serializzato alla memoria condivisa realizzato dal programmatore mediante un meccanismo di locking.
- La concorrenza basata sui processi (multiprocessing) si ha quando processi separati vengono eseguiti indipendentemente. I processi concorrenti tipicamente condividono i dati mediante IPC anche se possono usare anche la memoria condivisa se il linguaggio o la sua libreria la supportano.

12

Concorrenza in Python

- Python supporta sia la concorrenza basata sui thread che quella basata sui processi.
  - L'approccio al threading è alquanto convenzionale
  - L'approccio al multiprocessing è molto più ad alto livello di quello fornito da altri linguaggi. Il supporto al multiprocessing utilizza le stesse astrazioni del threading per facilitare il passaggio tra i due approcci, almeno quando non viene usata la memoria condivisa.

13

Problematiche legate a GIL

- Il Python Global Interpreter Lock (GIL) impedisce al codice di essere eseguito su più di un core alla volta
  - Si tratta di un lock che permette ad un solo thread di avere il controllo dell'interprete Python.
- Il GIL ha generato il mito che in Python non si può usare il mutithreading o avere vantaggio da un'architettura multi-core.

14

Concorrenza in Python

- In generale, se la computazione è CPU-bound, l'uso del threading può facilmente portare a perfomance peggiori rispetto a quelle in cui non si fa uso della concorrenza.
  - Una soluzione consiste nell'usare Cython che è essenzialmente Python con dei costrutti sintattici aggiuntivi che vengono compilati in puro C . Ciò può portare a performance 100 volte migliori più spesso di quanto accada usando qualsiasi tipo di concorrenza, in cui le performance dipendono dal numero di processori usati.
  - Se la concorrenza è invece la scelta più appropriata allora per evitare il GIL sarà meglio usare il modulo per il multiprocessing. Se usiamo il multiprocessing invece di usare thread separati nello stesso processo che quindi si contendono il GIL abbiamo processi separati che usano ciascuno la propria istanza dell'interprete Python senza bisogno di competere tra di loro.

15

Concorrenza in Python

- Se la computazione è I/O-bound, come ad esempio nelle reti, usare la concorrenza può portare a miglioramenti delle performance molto significativi.
- In questi casi i tempi di latenza della rete sono un tale fattore dominante che non ha importanza quale tipo di concorrenza utilizziamo.

16

Concorrenza in Python

- È raccomandabile scrivere prima la versione non concorrente del programma, se possibile.
  - Il programma non concorrente è più semplice da scrivere e da testare.
- Solo nel caso in cui questa versione del codice non fosse abbastanza veloce, si potrebbe scrivere la versione concorrente per fare un confronto sia in termini di correttezza che di performance.
- La raccomandazione è di usare il multiprocessing nel caso di computazione CPU-bound e uno qualsiasi tra multiprocessing e theading nel caso di programmi I/O bound.
- Oltre al tipo di concorrenza, è importante anche il livello di concorrenza.

Livelli di concorrenza

- **Low-Level Concurrency:** A questo livello di concorrenza si fa uso esplicito di operazioni atomiche (un'operazione atomica è un'operazione indivisibile che viene eseguita indipendentemente da qualsiasi altro processo, ovvero nessun'altra istruzione può cominciare prima che sia finita). Questo tipo di concorrenza è più adatta a scrivere librerie che a sviluppare applicazioni, in quanto può portare ad errori e rende difficile il debugging. Python non supporta questo livello di concorrenza anche se in esso la concorrenza è tipicamente implementata con operazioni di basso livello.
- **Mid-Level Concurrency:** Questo tipo di concorrenza non fa uso di operazioni atomiche esplicite ma fa uso di lock espliciti. Questo è il livello di concorrenza supportato dalla maggior parte dei linguaggi. Python fornisce il supporto per questo livello di concorrenza con classi quali **threading.Semaphore**, **threading.Lock** e **multiprocessing.Lock**. Questo livello di concorrenza è spesso usato per lo sviluppo di applicazioni perché spesso è l'unico disponibile.
- **High-Level Concurrency:** Questo livello di concorrenza non fa uso né di operazioni atomiche esplicite né di lock espliciti. Alcuni linguaggi stanno cominciando a supportare questo tipo di concorrenza. Python fornisce il modulo **concurrent.futures** e le classi **queue.Queue, multiprocessing.queue o multiprocessing.JoinableQueue** per supportare la concorrenza ad alto livello.

Dati modificabili condivisi

- Il problema chiave è la condivisione dei dati
  - Dati modificabili (mutable) condivisi devono essere protetti da lock per assicurare che tutti gli accessi siano serializzati in modo che un solo thread o processo alla volta possa accedere ai dati condivisi
  - Quando thread o processi multipli provano ad accedere agli stessi dati condivisi allora tutti ad eccezione di uno vengono bloccati. Ciò significa che quando viene posto un lock, la nostra applicazione può usare un unico thread o processo come se fosse non concorrente. Di conseguenza, è bene usare i lock il meno frequentemente possibile e per il più breve tempo possibile.

Livelli di concorrenza

- La soluzione più semplice consisterebbe nel non condividere dati modificabili. In questo modo non vi sarebbe bisogno di lock espliciti e non vi sarebbero problemi di concorrenza.
- A volte, thread o processi multipli hanno bisogno di accedere agli stessi dati ma ciò può essere risolto senza lock espliciti.
  - Una soluzione consiste nell'usare una struttura dati che supporta l'accesso concorrente. Il **modulo queue** fornisce **diverse code thread-safe**. Per la concorrenza basata sul multiprocessing possiamo usare le classi **multiprocessing.JoinableQueue** and **multiprocessing.Queue**.
    - La code forniscono una singola sorgente di job per tutti i thread e tutti i processi, e una singola destinazione dei risultati.
  - Alternative: dati non modificabili, deep copy dei dati e, per il multiprocessing, tipi che supportano l'accesso concorrente, come multiprocessing.Value per un singolo valore modificabile o multiprocessing.Array per un array di valori modificabili.

Informazioni sul pacchetto multiprocessing

- Un oggetto multiprocessing.Process rappresenta un'attività che è svolta in un processo separato. I metodi principali della classe sono:
- run(): metodo che rappresenta l'attività del processo
  - Può essere sovrascritto. Il metodo standard invoca l'oggetto callable passato al costruttore di Process con gli argomenti presi dagli argomenti args e kwargs, passati anch'essi al costruttore (si veda la prossima slide)
- start(): metodo che dà inizio all'attività del processo.
  - Deve essere invocato al più una volta per un oggetto processo.
  - Fa in modo che il metodo run() dell'oggetto venga invocato in un processo separato.
- join(*timeout*): Se l'argomento opzionale timeout è None (valore di default), il metodo si blocca fino a quando l'oggetto processo il cui metodo join() è stato invocato non termina. Se timeout è un numero positivo, join si blocca per al più timeout secondi. Il metodo resituisce None se il processo termina o se scade il tempo indicato da timeout.
  - Il metodo può essere invocato più volte per uno stesso oggetto processo.
  - un processo non può invocare join() su se stesso in quanto ciò provocherebbe un deadlock.

21

Informazioni sul pacchetto multiprocessing

- multiprocessing.Process(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs={}*, ***, *daemon =None*) deve essere sempre invocato con argomenti keyword:
  - group deve essere sempre None in quanto è presente solo per ragioni di compatibilità con threading.Thread di cui multiprocessing.Process condivide l'interfaccia
  - *target* è l'oggetto callable invocato da run(). Se è None vuol dire che non viene invocato alcun metodo.
  - name è il nome del processo
  - args è la tupla di argomenti da passare a target
  - *kwargs è un dizionario di argomenti keyword da passare a target*
  - daemon serve a settare il flag daemon a True o False. Il valore di default è None. Se daemon è None il valore del flag daemon è ereditato dal processo che invoca il costruttore.

per default non vengono passati argomenti a target.

22

Informazioni sul pacchetto multiprocessing

- I due metodi piu` usati per dare inizio ad un processo sono i seguenti:
  - *spawn:* Il processo padre lancia un nuovo processo per eseguire l'interprete python. Il processo figlio erediterà solo le risorse necessarie per eseguire il medodo run() degli oggetti processi. Questo modo di iniziare i processi è molto lento se confrontato con fork.
    - spawn è disponibile sia su Unix che su Windows. E` il default su Windows.
  - *fork:* Il processo padre usa os.fork() per fare il fork dell'interprete Python. Il processo figlio in questo caso è effettivamente identico al padre. Tutte le risorse sono ereditate dal padre.
    - Disponibile solo su Unix dove rappresenta il medodo di default per iniziare i processi.

23

Informazioni sul pacchetto multiprocessing

multiprocessing.Queue([*maxsize]*)

- restituisce una coda condivisa da processi
- maxsize è opzionale e serve a limitare il numero massimo di elementi che possono essere inseriti

- Metodi principali:
- qsize(): restituisce la dimensione approssimata della coda. Questo numero non è attendibile per via della semantica del multithreading/multiprocessing.
- empty(): restituisce True se la coda è vuota; False altrimenti. Anche l'output di questo metodo non è attendibile.
- full(): restituisce True se la coda è piena; False altrimenti. Anche l'output di questo metodo non è attendibile.

24

Informazioni sul pacchetto multiprocessing

- put(*obj*, *block*, *timeout*): inserisce obj nella coda. Se l'argomento opzionale block è True (default) e timeout è None (default), si blocca fino a che non si rende disponibile uno slot. Se timeout è un numero positivo, si blocca per al più timeout secondi e lancia l'eccezione queue.Full se non si rende disponibile nessuno slot entro quel lasso di tempo. Se block è falso, l'elemento viene inserito se è immediatamente disponibile uno slot altrimenti viene subito lanciata queue.Full (timeout viene ignorato).
- put_nowait(*obj*): equivalente a put(obj, False).
- get(*block*, *timeout*): rimuove e restitusce un elemento dalla coda. Se l'argomento opzionale block è True (default) e Timeout è None (default), si blocca fino a che un elemento è disponibile. Se timeout è un numero positivo si blocca per al più timeout secondi e lancia l'eccezione queue.Empty se nessun elemento si è reso disponibile in quel lasso di tempo. Se block è falso, viene restituito un elemento se ce ne è uno immediatamente disponibile, altrimenti viene subito lanciata queue.Empty (timeout viene ignorato).
- get_nowait(): equivalente a get(False).

25

Informazioni sul pacchetto multiprocessing

class multiprocessing.JoinableQueue

- JoinableQueue è una sottoclasse di Queue che ha in aggiunta i metodi task_done() e join().
- task_done() indica che un task precedentemente inserito in coda è stato completato. Questo metodo è usato dai fruitori della coda.
  - Per ciascuna get() usata per prelevare un task, deve essere effettuata una successiva chiamata a task_done per informare la coda che il task è stato completato.
  - Un join() bloccato si sblocca quando tutti i task sono stati completati e cioè dopo che è stata ricevuta una chiamata a task_done() per ogni item precedentemente inserito in coda.
  - Si ha un ValueError se task_done() è invocato un numero di volte maggiore degli elementi in coda.
- join() causa un blocco fino a quando gli elementi nella coda non sono stati tutti prelevati e processati.
  - Il conteggio dei task incompleti è incrementato ogni volta che viene aggiunto un elemento alla coda e viene decrementato ogni volta che un fruitore della coda invoca task_done() (se task_done() non fosse invocato ogni volta si potrebbe avere un overflow nel conteggio).
  - Quando il conteggio dei task va a zero, join() si sblocca.

26

Concorrenza ad alto livello: un esempio

- Supponiamo di voler scalare un insieme di immagini e di volerlo fare quanto più velocemente è possibile utilizzando più core.
- Scalare immagini è CPU-bound e quindi ci si aspetta migliori performance dal multiprocessing

Program	Concurrency	Seconds	Speedup
`imagescale-s.py`	*None*	784	*Baseline*
`imagescale-c.py`	4 coroutines	781	1.00×
`imagescale-t.py`	4 threads using a thread pool	1339	0.59×
`imagescale-q-m.py`	4 processes using a queue	206	3.81×
`imagescale-m.py`	4 processes using a process pool	201	3.90×

Tempi di esecuzione per processare 56 immagini su una macchina quad-coreAMD64 3GHz . La dimensione delle immagini va da 1MiB a 12MiB per un totale di 316MiB. L'output cnsiste di 67MiB.

27

Concorrenza ad alto livello: un esempio

- Il programmi per scalare immagini accetta i seguenti argomenti dalla linea di comando:
  - la dimensione a cui scalare le immagini
  - opzione se scalare o meno in modo smooth
  - directory delle immagini sorgente
  - directory delle immagini ottenute
- Immagini più piccole della dimensione indicata vengono copiate invece che scalate.
- Per le versioni concorrenti è anche possibile specificare la concorrenza (quanti thread o proccessi usare)
  - Per i programmi CPU-bound, normalmente usiamo tanti thread o processi quanti sono i core.
  - Per programmi I/O-bound, usiamo un certo multiplo del numero di core (2 ×, 3 ×, 4 ×, o di più) in base alla larghezza della banda della rete.

28

Concorrenza ad alto livello: un esempio

- La tupla di nome Result memorizza il conteggio di quante immagini sono state copiate e quante scalate che può essere (1,0) o (0,1) e il nome dell'immagine creata
- La tupla di nome Summary è usata per immagazzinare una sintesi di tutti i risultati.

```
Result = collections.namedtuple("Result", "copied scaled name")
Summary = collections.namedtuple("Summary", "todo copied scaled canceled")
```

- collections.namedtuple(*typename*, *field_names*, ***, *rename=False*, *defaults=None*, *module=None*) restituisce una nuova sottoclasse di tuple di nome typename.
- field_names è una sequenza di stringhe come ['x', 'y'] o può essere una singola stringa con ciascun nome separato da uno spazio e/o una virgola, come ad esempio 'x y' oppure 'x, y'.
  - per i nomi dei campi possono usati quelli ammessi per gli identificatori, ad eccezione dei nomi che cominciano con '_'
- La nuova sottoclasse è usata per creare tuple i cui campi sono accessibili come attributi, oltre ad essere indicizzabili e iterabili.
- Le istanze della sottoclasse hanno anche una docstring con typename e field_names e un utile metodo __repr__() che elenca il contenuto della tupla in formato name=value.
- Per gli altri argomenti si veda la documentazione.

29

Concorrenza ad alto livello: un esempio

- La funzione main legge la linea di comando con handle_commandline() che restituisce
  - la dimensione a cui occorre scalare l'immagine
  - un Booleano che indica se occorre usare uno scaling smooth
  - la directory sorgente da cui leggere le immagini
  - la directory destinazione dove scrivere le immagini ottenute
  - per le versioni concorrenti, il numero di thread o processori da utilizzare che per default è il numero di core.
- La funzione main riporta all'utente (con la funzione Qtrac.report()) che sta per eseguire la funzione scale() che è la funzione che svolge tutto il lavoro.
- Quando la funzione scale() restituisce la sintesi dei risultati, questa viene stampata usando la funzione summarize().

```
def main():
    size, smooth, source, target, concurrency = handle_commandline()
    Qtrac.report("starting...")
    summary = scale(size, smooth, source, target, concurrency)
    summarize(summary, concurrency)
```


30

Concorrenza ad alto livello: un esempio

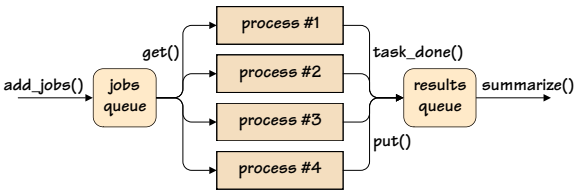
- La funzione scale() è il cuore del programma concorrente basato sulla coda.
- La funzione comincia creando una coda joinable di job da eseguire e una coda non joinable di risultati.
- Poi crea i processi per svolgere il lavoro e aggiunge job alla coda dei job con add_jobs()

```
def scale(size, smooth, source, target, concurrency):
    canceled = False
    jobs = multiprocessing.JoinableQueue()
    results = multiprocessing.Queue()
    create_processes(size, smooth, jobs, results, concurrency)
    todo = add_jobs(source, target, jobs)
    try:
        jobs.join()
    except KeyboardInterrupt: # May not work on Windows
        Qtrac.report("canceling...")
        canceled = True
    copied = scaled = 0
    while not results.empty(): # Safe because all jobs have finished
        result = results.get_nowait()
        copied += result.copied
        scaled += result.scaled
    return Summary(todo, copied, scaled, canceled)
```


31

Concorrenza ad alto livello: un esempio

- Con tutti i job nella coda dei job, si aspetta che la coda dei job diventi vuota usando il metodo multiprocessing.JoinableQueue.join().
  - Ciò avviene in un blocco a try ... except in modo che se l'utente cancella l'esecuzione (ad esempio, digitando Ctrl+C su Unix), possiamo gestire la cancellazione.
- Quando i job sono stati tutti eseguiti o il programma è stato cancellato, iteriamo sulla coda dei risultati.
  - Di solito, usare il metodo empty() su una coda concorrente non è affidabile ma qui funziona bene siccome tutti i processi worker sono terminati e la coda non viene più aggiornata.
  - Per questo stesso motivo possiamo usare il metodo multiprocessing.Queue.get_nowait() che non blocca gli altri processi invece del metodo multiprocessing.Queue.get() che invece blocca gli altri processi.



32

Concorrenza ad alto livello: un esempio

- Una volta accumulati i risultati, la tupla Summary viene restituita.
- In un'esecuzione normale, il valore todo è zero e cancelled è False; per un'esecuzione cancellata, todo è probabilmente non zero e cancelled è True.

```
def scale(size, smooth, source, target, concurrency):
    canceled = False
    jobs = multiprocessing.JoinableQueue()
    results = multiprocessing.Queue()
    create_processes(size, smooth, jobs, results, concurrency)
    todo = add_jobs(source, target, jobs)
    try:
        jobs.join()
    except KeyboardInterrupt: # May not work on Windows
        Qtrac.report("canceling...")
        canceled = True
    copied = scaled = 0
    while not results.empty(): # Safe because all jobs have finished
        result = results.get_nowait()
        copied += result.copied
        scaled += result.scaled
    return Summary(todo, copied, scaled, canceled)
```

33

Concorrenza ad alto livello: un esempio

- Questa funzione crea i processi per svolgere il lavoro.
- I processi ricevono la stessa funzione worker() (in quanto fanno tutti lo stesso lavoro) e i dettagli del lavoro che devono svolgere.
  - Ciò include la coda dei job condivisi e la coda dei risultati. Di norma non occorre mettere un lock a queste code condivise dal momento che le code stesse si occupano della loro sincronizzazione.
- Una volta creato un processo, esso viene trasformato in daemon in modo che termini nel momento in cui termina il processo principale. I processi non daemon continuano ad essere eseguiti anche una volta che è terminato il processo principale e su Unix diventano zombie.
- Dopo aver creato ciascun processo e averlo trasformato in daemon gli viene indicato di cominciare a svolgere la funzione che gli è stata assegnata. A quel punto ovviamente il daemon si blocca in quanto non abbiamo ancora inserito alcun job nella coda dei job.
  - Ciò non è importante dal momento che il blocco avviene in un processo separato e non blocca il processo principale. Di conseguenza, tutti i processi vengono creati velocemente e poi la funzione termina. Poi aggiungiamo job alla coda dei job per permettere ai processi bloccati di lavorare per eseguire questi job.

```
def create_processes(size, smooth, jobs, results, concurrency):
    for _ in range(concurrency):
        process = multiprocessing.Process(target=worker, args=(size,
                smooth, jobs, results))
        process.daemon = True
        process.start()
```

34

Concorrenza ad alto livello: un esempio

- Il codice proposto crea una funzione (worker) che viene passata come argomento (target) a multiprocessing.Process.
- La funzione worker esegue un loop infinito e in ogni iterazione prova a recuperare un job da svolgere dalla coda dei job condivisi. E` safe utilizzare un loop infinito in quanto il processo è un daemon e quindi terminerà al termine del programma.

```
def worker(size, smooth, jobs, results):
    while True:
        try:
            sourceImage, targetImage = jobs.get()
            try:
                result = scale_one(size, smooth, sourceImage, targetImage)
                Qtrac.report("{} {}".format("copied" if result.copied else
                        "scaled", os.path.basename(result.name)))
                results.put(result)
            except Image.Error as err:
                Qtrac.report(str(err), True)
        finally:
            jobs.task_done()
```

35

Concorrenza ad alto livello: un esempio

- Il metodo multiprocessing.get() si blocca fino a che non è in grado di restituire un job che in questo esempio è una tupla di due elementi, il nome della sorgente e il nome del target.
- Una volta recuperato il job, viene effettuato lo scale usando la funzione scale_one() e viene riportato ciò che è stato fatto. Viene anche inserito il risultato nella coda condivisa dei risultati

```
def worker(size, smooth, jobs, results):
    while True:
        try:
            sourceImage, targetImage = jobs.get()
            try:
                result = scale_one(size, smooth, sourceImage, targetImage)
                Qtrac.report("{} {}".format("copied" if result.copied else
                        "scaled", os.path.basename(result.name)))
                results.put(result)
            except Image.Error as err:
                Qtrac.report(str(err), True)
        finally:
            jobs.task_done()
```

36

Concorrenza ad alto livello: un esempio

- Una volta che sono stati creati e iniziati i processi, essi sono tutti bloccati nell'attesa di riuscire a prelevare job dalla coda dei job condivisi.
- Per ogni immagine da elaborare, questa funzione crea due stringhe: sourceImage che contiene l'intero percorso dell'immagine sorgente e targetImage che contiene l'intero percorso dell'immagine destinazione.
  - Ciascuna coppia di questi percorsi è aggiunta come tupla alla coda dei job. Alla fine la funzione restituisce il numero totale di job che devono essere svolti.
- Non appena il primo job è aggiunto alla coda, uno dei processi worker bloccati lo preleva e comincia a svolgerlo. La stessa cosa avviene per gli altri job inseriti fino a quando tutti i worker acquisiscono un job da svolgere. Più in là , è probabile che altri job vengano inseriti in coda mentre i processi worker stanno lavorando sui job prelevati. Questi nuovi job saranno prelevati non appena i worker fiscono di svolgere i job prelevati in precedenza. Quando i job nella coda terminano, i worker si bloccano in attesa di nuovo lavoro.

```
def add_jobs(source, target, jobs):
    for todo, name in enumerate(os.listdir(source), start=1):
        sourceImage = os.path.join(source, name)
        targetImage = os.path.join(target, name)
        jobs.put((sourceImage, targetImage))
    return todo
```

37

Concorrenza ad alto livello: un esempio

- Questa funzione è dove viene effettuato realmente lo scaling.
- Essa usa il modulo cyImage o il modulo Image se cyImage non è disponibile.
- Se l'immagine è già più piccola della dimensione data allora l'immagine viene semplicemente salvata nel file la cui path è specificata da targetImage. Viene quindi restituito Result per indicare che un'immagine è stata copiata e che nessuna è stata scalata e per specificare il file dell'immagine target.
- Altrimenti l'immagine è scalata e l'immagine risultante salvata. In questo caso il risultato Result informa che nessuna immagine è stata salvata e che una è stata scalata e indica il file dell'immagine target.

```
def scale_one(size, smooth, sourceImage, targetImage):
    oldImage = Image.from_file(sourceImage)
    if oldImage.width <= size and oldImage.height <= size:
        oldImage.save(targetImage)
        return Result(1, 0, targetImage)
    else:
        if smooth:
            scale = min(size / oldImage.width, size / oldImage.height)
            newImage = oldImage.scale(scale)
        else:
            stride = int(math.ceil(max(oldImage.width / size,
                                       oldImage.height / size)))
            newImage = oldImage.subsample(stride)
        newImage.save(targetImage)
        return Result(0, 1, targetImage)
```

38

Concorrenza ad alto livello: un esempio

- Una volta che tutte le immagini sono state processate, la funzione scale() crea e restituisce Summary che nel main viene passato alla funzione summarize.

```
def summarize(summary, concurrency):
    message = "copied {} scaled {} ".format(summary.copied, summary.scaled)
    difference = summary.todo - (summary.copied + summary.scaled)
    if difference:
        message += "skipped {} ".format(difference)
    message += "using {} processes".format(concurrency)
    if summary.canceled:
        message += " [canceled]"
    Qtrac.report(message)
    print()
```

- Una tipica sintesi prodotta da summarize è mostrata nella seguente figura

```
copied 0 scaled 56 using 4 processes
```