Concurrency

November 10, 2020

1 Concurrency and Parallelism

1.1 Concurrency

The main limitation to Python's concurrent execution is the **Global Interpreter Lock (GIL)**.

The *GIL* is a mutex that allows only **one thread** to run **at a given time** (per interpreter).

It is meant to patch *CPython* 's memory management, which is, in fact, a non-thread-safe reference counting.

While IO-bound threads are not affected by this limitation, CPU-bound threads are.

Python 3.8 should have brought some mitigations to this problem, but in practice nothing changes for the user.

python's standard libraries include:

- ***threading***: thread-based concurrency
- ***multiprocessing***: process-based parallelism
- ***concurrent.futures***: asynchronous execution via threads or processes [not covered in this lecture]

There are also external libraries that allow parallelism (e.g., *pathos*)

1.2 Parallelism

Parallelism in *python* is mainly used at data level. In fact, being the data independent, no synchronization is usually required.

This means that you can achieve full parallelism and take advantage of all of the cores in modern machines, squeezing all of their power.

After the processing, results can be handled by the parallel process or collected in the main process and handled.

So, how do we achieve parallelism in *python* ?

There are a couple of ways, and the right solution depends on your problem. For instance, you can:

- Spawn new processes, each with their own interpreter. This introduces non-negligible time and memory overhead
- Offload to external code. From a *python* perspective, this transforms a CPU-bound task to an IO-bound one. For example, *numpy* and other libraries implement most algorithms in C/C++/Fortran

1.2.1 *multiprocessing* package

Native parallelism is provided by the ***multiprocessing*** package.

The main building block of the package is the ***Process*** class, which instances represent the activity that is being run in a separate process. If you're familiar with *Java, it is quite similar to fairly to the* Java*'s equivalent.

Process init parameters include:

- ***target*** function to execute plus its ***args*** and ***kwargs***
- ***daemon***: whether the process is a daemon. They are killed when parent is closed

While the class methods include:

- ***run()***: the default invokes the target with its parameters. Can be overridden
- ***start()***: creates process and starts invokes the run method from there
- ***join()***: joins the process, with an optional timeout
- ***close()***: closes the Process instance and deallocates its resources

Example: *Process* init

[16]:
```python
from multiprocessing import Process

def f(name):
    print('hello', name)

# Process wants a TUPLE as args!
p = Process(target=f, args=('bob',))
print("INITED", p)
p.start()
print("STARTED", p)
p.join()
print("JOINED", p)
p.close()
print("CLOSED", p)
```

```
INITED <Process(Process-19, initial)>
hello bob
STARTED <Process(Process-19, started)>
JOINED <Process(Process-19, stopped)>
CLOSED <Process(Process-19, closed)>
```

multiprocessing* vs *multiprocessing.dummy The subpackage ***multiprocessing.dummy*** implements the same interface as the main package but is thread-based (i.e., logical concurrency but no parallelism, often used during testing).

As an example, let's start a *Process* from the two packages.

[2]:
```python
import multiprocessing
import multiprocessing.dummy as multithreading
```

```python
p = multiprocessing.Process()
t = multithreading.Process()

print(p)
print(t)
```

```
<Process(Process-2, initial)>
<DummyProcess(Thread-4, initial)>
```

As you can see, they offer the same interface but the result is different.

1.3 Ways to start a process

Processes can be started in three ways:

- ***fork***: forks the current *python* interpreter. It is available on Unix systems only, where it is the default method
- ***forkserver***: a server process is created and will create new processes on behalf of the parent. It is available on some Unix platforms;
- ***spawn***: a fresh python interpreter process is created. It inherits only the necessary resources to run the *Process* instance's *run()* method. This option can be faster or slower compared to the others as you need to reload some or all of the packages from disk. It is available on Unix and Windows, where it is the default option

The preferred method can be chosen using the *set_start_method(spawn_method)* function available in the *multiprocessing* package.

1.4 Synchronization

Synchronization between processes (or threads!) is, again, similar to *Java*. For instance, the *multiprocessing* package includes:

- Locks
  - ***Lock***: non-recursive lock. Subsequent acquisition attempts will block until the lock is released; any process or thread may release it
  - ***RLock***: recursive lock. The same process or thread may acquire it again and must release it the same number of times
- Semaphores
  - ***Semaphore***: atomic counter representing the number of *release()* calls minus the number of *acquire()* calls, plus an initial value. Can be acquired if the value is > 0
  - ***BoundedSemaphore***: like a *Semaphore*, but the counter cannot exceed its initial value

1.4.1 Examples

Locks What follows is a toy-example of acquiring a lock.

[17]:
```python
def f(lock):
    # We import here the resources to support all the spawn methods
    from time import sleep
    import multiprocessing
```

```python
    # We try to acquire the lock
    try:
        lock.acquire()

        print('{} says hello!'.format(multiprocessing.current_process()))

        # and sleep 3s after acquiring it.
        sleep(3)
    except Exception as e:
        print(e)
    finally:
        lock.release()
```

[18]:
```python
from multiprocessing import Lock, Process

# Get the lock instance
lock = Lock()

# Spawn two processes with sharing the lock
Process(target=f, args=(lock,)).start()
Process(target=f, args=(lock,)).start()
```

```
<Process(Process-20, started)> says hello!
<Process(Process-21, started)> says hello!
```

Semaphores *Semaphores* offer the same interface but different behaviour.

[5]:
```python
from multiprocessing import BoundedSemaphore, Process

# Init a semaphore with counter 2
semaphore = BoundedSemaphore(2)

for i in range(4):
    Process(target=f, args=(semaphore,)).start()
```

```
<Process(Process-5, started)> says hello!
<Process(Process-6, started)> says hello!
<Process(Process-7, started)> says hello!
<Process(Process-8, started)> says hello!
```

1.5 Sharing objects

Python also supports sharing objects between processes (and threads).

1.5.1 Pipes

The most basic way is sharing using ***Pipe*** objects, although this solution is not very pythonic and more user friendly ways exist.

Pipes objects allow sending objects from one end to the other.

They may be duplex (send and receive from both sides) or not.

Note that they can get corrupted if two or more processes/threads read from or write to from the same side.

[6]:
```python
from multiprocessing import Pipe, Process
from time import sleep

def send_something(conn):
    display("Hello!")

    sleep(1)

    conn.send([42, None, 'hello'])
    conn.close()

# Monoplex Pipe. The first end can only receive, the other can only send
conn_receive, conn_send = Pipe(duplex=True)

# Init process
p = Process(target=send_something, args=(conn_send,))

print("Starting the process")
p.start()

print("Waiting for a message")
# Wait to receive something
print("Received:", conn_receive.recv())
```

```
Starting the process
Waiting for a message
Received: [42, None, 'hello']
```

1.5.2 Queues

For instance, the *multiprocessing* package also includes various **queue** implementations. They all allow to define the max queue size (0 <= means infinite) and support different in-out policies.

The most used implementations are: - ***Queue***: FIFO queue - ***LifoQueue***: LIFO queue - ***PriorityQueue***: priority queue

[7]:
```python
from multiprocessing import Queue, Process

def producer(queue):
```

```
    from time import sleep

    for i in range(10):
        queue.put(i)

        sleep(1)

    queue.put(None)


def consumer(queue):
    while True:
        item = queue.get()

        if item is None:
            break

        print(item)
```

[8]:
```
from multiprocessing.dummy import Process as Thread

queue = Queue()
p = Process(target=producer, args=(queue,))
p.start()

t = Thread(target=consumer, args=(queue,))
t.start()

# t.join()
# p.join()
```

```
0
```

What if we use two daemons instead?

[9]:
```
p = Process(target=producer, args=(queue,), daemon=True)
p.start()

t = Process(target=consumer, args=(queue,), daemon=True)
t.start()

# t.join()
# p.join()
```

```
0
1
2
3
```

```
4
5
6
7
8
9
```

Computation will not complete unless we join them!

1.6 Sharing state

Python supports sharing state between processes and threads.

Keep in mind that it is usually best to avoid using shared state as far as possible. - This is particularly true when using multiple processes

There are a couple of ways of sharing state: - Sharing memory - *Managers*

1.6.1 Shared memory

Starting from **Python 3.8**, you can also share any object using the *shared_memory* module.

It allows to share a location of memory between processes (threads already share memory, of course) and allocate base objects there.

Very briefly, you can allocate *CTypes* object in a shared memory:

- ***Value*** represents a single value
- ***Array*** represents an array

Check the docs for more!

[10]:
```python
from multiprocessing import Process, Value, Array, RLock

def shared_memory_consumer(shared_value, array):
     with shared_value.get_lock():
        shared_value.value = 10

     with array.get_lock():
        for i in range(len(array)):
            array[i] = len(array) - i

# Init value as double (float), protected by a lock
shared_value = Value("d", 0.0, lock=True)

# Init array of integers, protected by a REENTRANT LOCK
array = Array('i', range(10), lock=RLock())

p = Process(target=shared_memory_consumer, args=(shared_value, array))
p.start()
p.join()
```

```python
print(shared_value.value)
print([array[i] for i in range(len(array))])
```

```
10.0
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

1.6.2 *Managers*

Managers provide a way to create data which can be shared between different processes.

They **control a server** process holding the objects and allows other processes to manipulate them using **proxies**.

including sharing over a network between processes running on different machines.

For instance, the ***SyncManager***, returned by ***Manager()***, supports lists, dictionaries, locks, semaphores, queues, shared memory objects and others.

[11]:
```python
from multiprocessing import Manager

manager = Manager()

lock = manager.Lock()
semaphore = manager.BoundedSemaphore()

queue = manager.Queue()

value = manager.Value("d", 0.0, lock=True)
```

1.7 Pools

The most common, but also simple and pythonic, way to perform multiprocessing in *python* is through **pools** of processes.

Pools create a number of workers which will carry out tasks submitted to the pool.

A ***Pool*** object controls a pool of workers, and supports both synchronous and asynchronous results.

1.7.1 *Pool* parameters

The main parameters of the ***Pool*** class include:

- ***processes***: number of worker processes to use. If *None*, the number of CPUs is used
- ***initializer***: if not *None*, each worker will call ***initializer(*initargs)*** when it starts
- ***maxtasksperchild***: number of tasks a worker can complete before it will exit and be replaced with a fresh worker process, to enable unused resources to be freed. Default is *None*, which means worker processes will live as the pool itself

Example: *Pool* initialization Let's init a *Pool*.

[12]:
```python
from multiprocessing import Pool

def square(x):
    return x*x

def square_wait(x):
    from time import sleep

    sleep(2)
    return x*x

pool = Pool(processes=4)

# pool = Pool()
print(multiprocessing.cpu_count())
```

```
8
```

1.7.2 *Pool* methods

Pool objects offer both **synchronous** and **asynchronous** methods.

Synchronous methods The *synchronous* methods are: - ***apply(func[, args[, kwds]])***: calls *func* with given arguments - ***map(func, iterable[, chunksize])***: chops the *iterable* parameter into chunks of (approximate) size *chunksize* and submits them to the pool as separate tasks - ***imap(func, iterable[, chunksize])***: *map* lazier variant. Suitable for very long iterables using a large chunksize value improves performances - ***imap_unordered(func, iterable[, chunksize])***: same as above, but results' order is arbitrary - ***starmap(func, iterable[, chunksize])***: like *map* but the elements of the *iterable* are expected to be iterables that are unpacked as arguments

Getting results synchronously After initing the *Pool*, let's submit a job and get the result synchronously.

[19]:
```python
result = pool.apply(square, (2,))
print("This result will show immediately. Result:", result)
result = pool.apply(square_wait, (2,))
print("This result will take some time. Result:",result)
```

```
This result will show immediately. Result: 4
This result will take some time. Result: 4
```

Asynchronous method The synchronous methods also have an **asynchronous** variant:

- ***apply_async(func[, args[, kwds[, callback[, error_callback]]]])***
- ***map_async(func, iterable[, chunksize[, callback[, error_callback]]])***
- ***starmap_async(func, iterable[, chunksize[, callback[, error_callback]]])***

While the synchronous result methods block until the result is ready, the asynchronous ones return an ***AsyncResult*** object and also provide timeouts and callbacks.

The *AsyncResult* provides blocking ***get()*** and ***wait()*** methods to get the result, and ***ready()*** and ***successful()*** methods to check the result status. The single argument callbacks can handle the result or the exception, but must return immediately as they are executed by the main thread and block the result processing otherwise.

Getting results asynchronously Now let's try to get the results asynchronously.

[14]:
```python
# Get the result asynchronously
result = pool.apply_async(square_wait, (5,))

print("Here's your (future) result", result)
print("WAIT FOR IT")
result.wait()
print("RESULT IS READY!")
print(result.get())
print("Status successful? {}".format(result.successful()))
```

```
Here's your (future) result <multiprocessing.pool.ApplyResult object at
0x7f79ca75def0>
WAIT FOR IT
3
4
RESULT IS READY!
25
Status successful? True
```

[15]:
```python
result = pool.apply_async(square_wait, (10,))

print("Timeout 1s, but function takes more time!")
print(result.get(timeout=1))
```

```
Timeout 1s, but function takes more time!
5
```

```
---------------------------------------------------------------------------
TimeoutError                              Traceback (most recent call last)
<ipython-input-15-8b3ab813e71f> in <module>
      2
      3 print("Timeout 1s, but function takes more time!")
----> 4 print(result.get(timeout=1))

~/anaconda3/envs/pytorch/lib/python3.7/multiprocessing/pool.py in get(self,␣
 ↪timeout)
    651         self.wait(timeout)
    652         if not self.ready():
--> 653             raise TimeoutError
    654         if self._success:
    655             return self._value
```

```
TimeoutError:
```

Map iterators to workers Let's try to map an iterator to many workers.

[20]:
```python
# Define a processor
def square_random_wait(x):
    from random import randint
    from time import sleep

    sleep_time = randint(0, 2)
#     print("Sleeping for", sleep_time, "seconds")

    sleep(sleep_time)

    return x*x


from multiprocessing import Pool

# Init our pool
pool = Pool()

# Define input values
values = range(4)
```

[21]:
```python
print("These results will appear immediately")
print(pool.map(square, values))
```

```
These results will appear immediately
[0, 1, 4, 9]
```

[22]:
```python
print("These results will appear all at once")
print(pool.map(square_random_wait, values))
```

```
These results will appear all at once
[0, 1, 4, 9]
```

[23]:
```python
print("These results will appear one at a time *IN THE SAME ORDER AS THE INPUT*!
 ↪")

for result in pool.imap(square_random_wait, values):
    print(result)
```

```
These results will appear one at a time *IN THE SAME ORDER AS THE INPUT*!
0
1
```

```
4
9
```

[24]:
```python
values = range(10)

print("These results will appear one at a time *AS SOON AS THEY ARE READY*!")
for result in pool.imap_unordered(square_random_wait, values):
    print(result)
```

```
These results will appear one at a time *AS SOON AS THEY ARE READY*!
0
16
25
4
49
1
81
9
36
64
```

1.7.3 *Pool* handling

Pools provide a few methods to handle them: - ***close()***: prevents any more tasks from being submitted. Once all the tasks have been completed the worker processes will exit - ***terminate()***: forces the workers to exit - ***join()***: waits for the worker processes to exit. Must be called **after** ***close()*** or ***terminate()***

The simplest way to handle a pool of workers is through the ***with*** statement (Context Manager protocol). - It automatically closes the pool once the *with* block is done. Behaves like a *try … finally* - Generic purpose, not limited to pools

[27]:
```python
from multiprocessing import Pool, current_process
from time import sleep

def random_wait(x):
    from random import randint
    sleep(randint(0, 2))

    print("{} DONE!".format(current_process()))


values = range(10)

with Pool(processes=4) as pool:
    pool.map_async(random_wait, values)

print("These tasks will not complete")
```

These tasks will not complete

```
[28]: with Pool(processes=4) as pool:
          pool.map_async(random_wait, values)

          pool.close()
          pool.join()

      print("But these will!")
      # print(results.get())
```

```
<ForkProcess(ForkPoolWorker-44, started daemon)> DONE!
<ForkProcess(ForkPoolWorker-44, started daemon)> DONE!
<ForkProcess(ForkPoolWorker-43, started daemon)> DONE!
<ForkProcess(ForkPoolWorker-42, started daemon)> DONE!
<ForkProcess(ForkPoolWorker-45, started daemon)> DONE!
<ForkProcess(ForkPoolWorker-44, started daemon)> DONE!
<ForkProcess(ForkPoolWorker-45, started daemon)> DONE!
<ForkProcess(ForkPoolWorker-44, started daemon)> DONE!
<ForkProcess(ForkPoolWorker-43, started daemon)> DONE!
<ForkProcess(ForkPoolWorker-42, started daemon)> DONE!
But these will!
```

1.8 Guidelines

Here are a few guidelines from the *python*'s official documentation that aim to improve your code and avoid bugs.

- **Avoid shared state**. Stick to queues or pipes rather than using the lower level synchronization primitives
- **Prefer inheritance than pickle/unpickle**
  - … and also **be sure** that your **arguments are** *picklable* (**serializable**)
- **Lock proxies if multi*threading***. They are **NOT *thread* safe**!
- **Explicitly pass resources to child** processes for compatibility with the spawn method, which is default on Windows, **instead** using **of global resources**
- **Do not terminate processes abruptly if they use shared resources**
- **Join processes that use queues carefully**. They wait before terminating until all the buffered items are fed to the underlying pipe and joining them will cause deadlocks

1.9 Exercise

Follow up of the previous section [**Iterables and generators**]({{< ref "iterables" >}}).

Now we want to process the records in the CSV file. Assume that you want to perform some very time consuming operation on them and employ a *Pool* of processes to perform these operations.

Note that *map* will unroll the generator and fit all the records into memory, which contrasts with our requirements. For a moment, forget about it and use *map_async*.

As a second step, reintroduce the RAM constraint and use queues and *apply_async*. A few tips for this second step: - Use *Queues* - This is just a producer/consumer example

[ ]:
```python
from pathlib2 import Path


def dataset_reader(file):
    # Lets use pathlib instead of using the open() function,
    # with open(file, "r+") as f:

    # Creating a Path instance.
    file = Path(file)

    # Not needed, just showing Pathlib off a bit
    if not file.absolute():
        file = file.resolve()

    with file.open("r+", encoding="ISO-8859-15") as f:
        header = f.readline()
        columns = header.strip().split(',')

#         print(columns)
        for line in f:
            values = line.strip().split(',')
#             print(values)

            try:
                yield dict(zip(columns, values))
            except GeneratorExit:
                print("Closing the generator!")
                break


file = "albumlist.csv"
```

[ ]:
```python
from tqdm import tqdm
from subprocess import check_output
from multiprocessing import Pool, current_process

# Redirect STDOUT to TQDM
def print(x):
    tqdm.write(str(x))

processes_num = 4

file = "albumlist.csv"
generator = dataset_reader(file)
```

```python
records_number = None
cmd = "wc --lines {}".format(Path(file).resolve())

try:
    records_number = check_output(cmd, shell=True, text=True)
    records_number = int(records_number.strip().split()[0])
except Exception as e:
    exit("ERROR! {}".format(e))

print("{} -> {}".format(cmd, records_number))
```

[ ]:
```python
# Let's define a very very complex record-processing function:

def f(x):
    return x["Number"]
```

1.9.1 Part 1: no memory constraints

[ ]:
```python
with Pool(processes=processes_num) as pool:
#     pool.map_async(lambda x: x["Number"], tqdm(generator,␣
↪total=records_number), callback=callback)

    pool.map_async(f, tqdm(list(generator), total=records_number),␣
↪callback=print)
    pool.close()
    pool.join()
```

1.9.2 Part 2: bring memory constraints back

[ ]:
```python
def producer(queue, generator):
    for record in generator:
        #print(record)
        queue.put(record)

    queue.put(None)

    print("PRODUCER DONE!")
    return


def consumer(queue, function):
    output = []
    while True:
        #print("get!")
        item = queue.get()
```

```
        if item is None:
            print("CONSUMER DONE!")
            break

        output.append(function(item))
        #print(item)

    print(output)
    return output
```

[ ]:
```
from multiprocessing import Manager, Queue
from multiprocessing.dummy import Process as Thread

manager = Manager()
queue = manager.Queue(maxsize=processes_num)


generator = dataset_reader(file)

with Pool(processes=processes_num) as pool,\
    tqdm(total=records_number) as progressbar:

    #Thread(target=producer, args=(queue, generator)).start()


    def callback(x):
        progressbar.update(len(x))

#         print(x)

    for _ in range(processes_num):
        pool.apply_async(func=consumer,
                         args=(queue, f),
                         callback=callback,
                         error_callback=print
                         )

    for record in generator:
#         print(record)
        queue.put(record)

    for _ in range(processes_num):
        queue.put(None)

    pool.close()
    pool.join()
```

1.10 References

- Python Docs: concurrency
- Python Docs: multiprocessing
- Brendan Fortuner at Medium
- Chriskiehl