UNIVERSITÀ
DEGLI STUDI
FIRENZE

Parallel Programming

Prof. Marco Bertini

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Python: multiprocessing

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Python and process-based parallelism

`multiprocessing` module

- The global interpreter lock (GIL) prevents more than one piece of Python bytecode from running concurrently.

- This means that for anything other than I/O bound tasks, excluding some small exceptions, using multithreading won't provide any performance benefits the way it would in C/C++ using OpenMP (and all the other languages and libraries like Java, C++11, Pthreads, etc.)

- The Python multiprocessing library, which is part of the standard library of the language, implements the shared memory programming paradigm and let us overcome the GIL limitation.

Processes vs. threads

- Multiprocessing let us spawn processes instead of creating threads.

  - Instead of our parent process spawning threads to parallelize things, we instead spawn subprocesses to handle our work.

  - The parent process can of course continue its execution asynchronously or wait until the child process ends its execution.

  - Each subprocess we spawn will have its own Python interpreter and its own GIL. These subprocesses can run on different cores, obtaining parallelism.

Spawning a process

- The multiprocessing library of Python allows the spawning of a process through the following steps:

  1. Build the object process.

  2. Call its `start()` method. This method starts the process's activity.

  3. Call its `join()` method. It waits until the process has completed its work and exited.

- Protect the creation of the processes in the main module (necessary on Windows, good practice on Linux), using:
  `if __name__ == "__main__"`

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Multiprocessing basics

Creating a process

- The Process class takes in two arguments, a `target` which is the function name we wish to run in the process and `args` which is a tuple of arguments we wish to pass to the function.

- It's possible to add a `name` that can be retrieved with `multiprocessing.current_process().name`

- Target functions can be imported

Starting/Joining a process

- The `start` method returns instantly and will start running the process.

- The `join` method of a process will cause our main process to block until each process has finished. It can receive a timeout params in seconds, after timeout expiration it returns.
  - Without this, a program would exit almost instantly and terminate the subprocesses as nothing would be waiting for their completion.

- Using this approach we don't get the return values of the functions executed in the processes !
  - Get the process exit code with `exitcode` attribute.
    0 == OK, >0 error, <0 killed

Daemon processes

- Running a process in background is a typical mode of execution of laborious processes that do not require your presence or intervention.

- Set the `daemon` attribute to `True` to execute a process in background

  - A deamon process is not a daemon server !

  - The daemon process is terminated automatically before the main program exits, which avoids leaving orphaned processes running.

Subclassing `Process`

- To implement a custom subclass and process:
  - Define a new subclass of the `Process` class
  - Override the `_init__(self [,args])` method to add additional arguments, if needed
    - Override the `run(self [,args])` method to implement what Process should when it is started
- Once the new Process subclass is created, create an instance of it and then start by invoking the `start()` method, which will in turn call the `run()` method.
  - Remind to `join()` the object.

Queues

- The multiprocessing library has two communication channels with which it can manage the exchange of objects: queues and pipes.

- A queue returns a process shared queue that is thread and process safe, multi-producer, multi-consumer FIFO.
  Any serializable object (i.e. pickable objects) can be exchanged through it.

- Instantiate in `__main__` and pass it to the processes (e.g. Process subclasses) to let them communicate (e.g. to implement producer/consumer).

Queues

- A queue has the `JoinableQueue` subclass. It should be used to send tasks to processes. It has two additional methods:

- `task_done()`: indicates that a task is complete, for example, after the `get()` method is used to fetch items from the queue. It must be used only by queue consumers.

- `join()`: blocks the processes until all the items in the queue have been taken and processed.

Pipes

- A pipe does the following:
  - Returns a pair of connection objects connected by a pipe
  - Every object has send/receive methods to communicate between processes

Shared memory

- Data can be stored in a shared memory map using multiprocessing `Value` or `Array`, and specifying the type of data stored (e.g. `'d'` for double and `'i'` for signed integer)
  - Deal with race conditions using synchronization, e.g. locks
- A `Manager` object creates a server process that allows other processes to manipulate Python objects

Synchronization

- `multiprocessing.Lock()` is a mutex that can be acquired/released using `acquire()`/ `release()` methods or using with

- `multiprocessing.Semaphore()` is a semaphore that limits the number of concurrent accesses to a resource and has a similar interface to Lock

- Also barrier and condition variables (to synchronize parts of a workflow), are available

Synchronization

- All the synchronization classes that have `acquire()` and `release()` methods can be used as context managers for a `with` statement. The `acquire()` method will be called when the block is entered, and `release()` will be called when the block is exited.

  - ```
    with some_lock:
        # do something...
    ```

  - is equivalent to:

  - ```
    some_lock.acquire()
    try:
        # do something...
    finally:
        some_lock.release()
    ```

Pool

- The `Pool` class can be used to manage a fixed number of workers for cases where the work to be done can be broken up and distributed between workers independently.

- The return values from the jobs are collected and returned as a list.

- The pool arguments include the number of processes and a function to run when starting the task process

Pool methods

- `apply()`: It blocks until the result of the passed function is ready.

- `apply_async()`: This is a variant of the `apply()` method, which returns a result object (`AsyncResult`). It is an asynchronous operation that will not lock the main thread until all the child classes are executed. Can receive callbacks for result, applied when it's ready.

- `map()`: This is the parallel equivalent of the `map()` built-in function that return an iterator that applies a function to every item of iterable, yielding the results. It blocks until the result is ready, this method chops the iterable data in a number of chunks that submits to the process pool as separate tasks.

- `map_async()`: This is a variant of the `map()` method, which returns a result object. If a callback is specified, then it should be callable, which accepts a single argument. When the result becomes ready, a callback is applied to it (unless the call failed). A callback should be completed immediately; otherwise, the thread that handles the results will get blocked.

- To close the `Pool` first call `close()` or `terminate()` methods, then `join()`

Pools and sharing data

- A task submitted to a process pool, may not run immediately because the processes in the pool may be busy with other tasks.
  - To deal with this, when we submit a task to the process pool, its arguments are pickled (serialized) and put on the task queue.
- Both `Value` and `Array` objects are not pickleable, so they can't be passed to processes in pools.
- We'll need to put our shared data in a global variable and to let our worker processes know about it we use *process pool initializers*, i.e. special functions that are called when each process in our pool starts up. Using this we can create a reference to the shared memory created in the parent process.

Futures

- Futures are provided in `concurrent.futures`, that provides thread and multiprocessing pool executors to schedule asynchronous callables.
  - Only multiprocessing pool executors provide parallelism… They are an abstraction over process pools.
  - The two pool executors share the same abstract `Executor` interface
    - We can `submit()` a function and get an associated future
    - We can `map()` a function to iterables, setting optional timeout and chunk size arguments. Similar to `Pool` this method chops iterables into a number of chunks which it submits to the pool as separate tasks. Increasing `chunksize` w.r.t. default 1 can improve performance.

Pool executors

- `submit()` which will take a callable and return a `Future` (note that this is not the same as `asyncio` futures, but is part of the `concurrent.futures` module) – is equivalent to the `Pool.apply_async` method

- `map()` which will take a callable and a list of function arguments and then execute each argument in the list asynchronously. It returns an iterator of the results of our calls similarly to `asyncio.as_completed` since results are available once they complete.

  - … but the order of iteration is deterministic based on the order we passed in the arguments. We can't get the following results until the previous one has returned ! Thus it's less responsive than `asyncio.as_completed`

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Multiprocessing and asyncio

Pool executors and asyncio

- Once we have a pool, we can use a special method on the `asyncio` event loop called `run_in_executor`.

- This method will take in a *callable* alongside an executor (using a process pool we have parallelism) and will run that callable inside the pool.

- It then returns an *awaitable* which we can use in an await statement or pass into an API function such as `gather` to wait for tasks completion.

run_in_executor

- `run_in_executor` only takes a callable and does not allow us to supply function arguments
  - to get around this we can use *partial function application*.
  - Partial application takes a function that accepts some arguments and turns it into a function that accepts fewer arguments. It does this by “freezing” some arguments that we supply. It is implemented in the `functools` module:

`foo(some_arg)` can be transformed in `foo_some_args()` using:

```
foo_some_args = functools.partial(foo,
                                  some_args)
```

Using executors with asyncio

1. create a process pool executor

2. get the `asyncio` event to be able to call `run_in_executors`

3. create partial functions

4. call `run_in_executor` for each partial function and keep track of the returns

5. wait on the returns using `gather` or use `asyncio.as_completed` to get results as they are available

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Joblib basics

What is Joblib ?

- `Joblib` is a high level library that provides:
  - capability to use cache, to avoid recomputation of some of the processing steps (memoization)
  - Executes parallelization using different backends (like `multiprocessing` or `loky`). Can use `threads` to deal with compiled extensions that do not use GIL. Can use `ray` for cluster processing.
  - More efficient persistenza than `pickle`
- Install with `pip` or `conda`. If `loky` backend is used install it as well.
- `Scikit-learn` uses Joblib for parallelization

Backend

- Using the `multiprocessing` backend means that no code should run outside of the `if __name__ == '__main__'` block (to deal with Windows)
  - Uses `multiprocessing.Pool`
- Using `loky` this is not required anymore
  - This is the suggested use.
  - Install loky using: `pip3 install loky`

Memoization

- Memoization is an optimization technique that stores the results of expensive function calls, returning the cached result when the same inputs occur again.

- Memoization lowers time cost in exchange for space cost; memoized functions are optimized for speed in exchange for a higher use of computer memory space.

- The Python Decorator Library has a decorator called `memoized` but it stores results in memory

- Joblib can store on disk using optimized compressed dumps. Can be used on numpy arrays.

  - Can be used as decorator or as `Memory` object method

Memory tricks

- Function cache is identified by the function's name. Thus assigning the same name to different functions, their cache will override each-others.

- `Memory` is designed for pure functions and it is not recommended to use it for methods. To use cache inside a class the recommended pattern is to cache a pure function and use the cached function inside the class.

- `clear()` method erases the cache directory. All data stored there will be lost !

Parallelization

- Joblib provides a simple helper class `Parallel` to write embarrassingly parallel for loops using multiprocessing.

- To share function definition across multiple python processes, a serialization protocol is used (based on `pickle`).

- `joblib` will run each function call in an isolated Python processes, therefore they cannot mutate a common Python object defined in the main program.

  - However if the parallel function needs to rely on the shared memory semantics of threads, it should be made explicit with `require='sharedmem'` to access a variable used in their body.

Parallelization

- It may be needed to call parallel functions interleaved with processing of the intermediate results.
- Calling `joblib.Parallel` several times in a loop is sub-optimal because it will create and destroy a pool of workers (threads or processes) several times which can cause a significant overhead.
- For this case it is more efficient to use the context manager API of the `joblib.Parallel` class (`with … as …:`) to re-use the same pool of workers for different consecutive calls.
  - Similar to the OpenMP shared `#pragma omp parallel` for multiple parallel directives

Parallelization

- The arguments passed as input to the `Parallel` call are serialized and reallocated in the memory of each worker process.
  - This can be problematic for large arguments as they will be reallocated many times.
- To deal with `numpy` data structures, Joblib provides a special handling to use references to the same filesystem object using `numpy.memmap` subclasses of `numpy.ndarray`
- It is possible storage location and a threshold size that triggers the automatic conversion of `numpy` arrays into `numpy.memmap` objects
- It is possible to manually manage memmap-ed data for finer optimization tuning of memory usage.

Parallelization

- Many HPC and scientific libraries manage their own thread pools, and used in conjunction with Joblib we may have too many threads (Joblib processes + library threads)

- It is possible to specify the maximum number of threads that (some supported libraries) can spawn through the `parallel_backend()` method

  - Joblib supports OpenMP, Intel MKL, OpenBLAS, etc.

Parallelization

- Use `Parallel` `verbose` argument for logging
- Use `n_jobs` argument to set max. Number of parallel jobs
- Use `delayed` decorator to capture the arguments of a parallelized function. It creates a tuple (function, args and kwargs) with a function call syntax and delays the execution of such tuple to allow to pass it to the `Parallel` object.
  - Without it the list of functions in the loop would return before the list is passed to the `Parallel` object and the execution would be sequential.

Serialization

- `joblib.dump()` and `joblib.load()` provide a replacement for pickle to work efficiently on arbitrary large Python objects, in particular large numpy arrays.
  - Alternatively Python >= 3.8 + numpy >= 1.16 allow to use `pickle protocol=5` that is efficient way to deal with large numpy arrays
- Joblib supports 10 compression levels (0=none), 3 is considered a good default
- Joblib supports ‘gzip’, ‘bz2’, ‘lzma’ and ‘xz’ compressions. Lz4 is available if the corresponding package is installed.
  - Select compression from the file extension or using the `compress` argument using a tuple (or set it to `True` for default zlib level 3 compression)

Books

- Python Concurrency with asyncio, M. Fowler, Manning - Chapt. 6

- Python Parallel Programming Cookbook, G. Zaccone, Packt Publishing, Chapt. 3

- The Python 3 Standard Library by Example, D. Hellmann, Addison Wesley, Chapt. 10