Multiprocessing.Pool() - Stuck in a Pickle

https://thelaziestprogrammer.com/python/a-multiprocessing-pool-pickle

The Author Wed Jul, 28 16:14

Intro

This post sheds light on a common pitfall of the Python `multiprocessing` module: spending too much time serializing and deserializing data before shuttling it to/from your child processes. I gave a talk on this blog post at the Boston Python User Group in August 2018

Generally speaking, concurrent programming is hard. Luckily for us, Python's `multiprocessing.Pool` abstraction makes the parallelization of certain problems extremely approachable.

```
from multiprocessing import Pool


def sqrt(x):
    return x**.5


numbers = [i for i in range(1000000)]
with Pool() as pool:
    sqrt_ls = pool.map(sqrt, numbers)
```

The basic idea is that given any iterable of type `Iterable[T]`, and any function `f(x: T) -> Any`, we can parallelize the higher-order function `map(f, iterable)` with 1 line of code. The above iterates over 1 million integers, and in parallel, calculates the `sqrt` of each integer, utilizing all CPUs on our machine.

That said, this post is about getting into trouble, not about the simple case above.

A More Complex Case: IntToBitarrayConverter

Our toy class will do just what it says it does: **convert** `ints` to their binary representation of **0**s and **1**s. Specifically, our implementation will return a bitarray as a `numpy.ndarray`. Our class also stores a cache of `int` -> `str`, implemented as a simple `dict`, which quickly converts `int` keys to `str` values (*bitstrings*).

```
class IntToBitarrayConverter():

    def set_bitstring_cache(self, bitstring_cache: Dict):
        self.bitstring_cache = bitstring_cache

    def convert(self, integer: int) -> np.ndarray:
        bitstring = self.bitstring_cache[integer]
        # cache the step of bitstring = format(integer, 'b')
        return self._bitstring_to_ndarray(bitstring)

    @staticmethod
    def _bitstring_to_ndarray(bitstring) -> np.ndarray:
        arr = (np.fromstring(bitstring, 'u1') - 48)
        return arr
```

Note: that `np.fromstring(bitstring, 'u1') - 48` parses the str as the 8-bit integer ASCII values of the '0' and '1' chars (48 and 49 respectively), and subtracts 48 to yield binary data.

We can convert 1000000 `ints` to their `np.ndarray` bitarray representations with the code below:

```
CACHE_SIZE = 1024 * 1024  # 40 MB
ITER_SIZE = 1000000  # 1 million

int_to_bitarr_converter = IntToBitarrayConverter()
int_to_bitarr_converter.set_bitstring_cache(
    {key: format(key, 'b') for key in range(CACHE_SIZE)})
ndarray_bitarr_ls = list(
    map(int_to_bitarr_converter.convert,
        (random.randint(0, CACHE_SIZE - 1)
         for _ in range(ITER_SIZE))))
```

Running the above on my 2017 Macbook Pro, I see a **wall time of 4.94s**. Five seconds is not an extraordinarily long span of time, but I am greedy, and I want to run this concurrently and speed things up. Fortunately, this is easy with our `Pool` abstraction:

```
from multiprocessing import Pool


CACHE_SIZE = 1024 * 1024  # 40 MB
ITER_SIZE = 1000000  # 1 million

int_to_bitarr_converter = IntToBitarrayConverter()
int_to_bitarr_converter.set_bitstring_cache(
    {key: format(key, 'b') for key in range(CACHE_SIZE)})
with Pool() as pool:
    ndarray_bitarr_ls = pool.map(
        int_to_bitarr_converter.convert,
        (random.randint(0, CACHE_SIZE - 1)
         for _ in range(ITER_SIZE))))
```

Running the above on the same machine, this takes **32.5s**. This a 600% **slow-down**! What is going on?

Stuck in a Pickle

[Link to Boston Python User Group Lightning Talk Diagram]

Under the hood, our call to `pool.map(...)` does the following:

1. Initializes 3 `Queues`:
    1. The `taskqueue` which holds `tuple` of `tasks`: `(result_job, func, (x,), {})`.
        1. We only care about `(x,)` above. This holds our function `convert()`, and a **chunk** of elements from our `iterable`.
    2. The `inqueue`, which holds serialized (*pickled*) `tasks`.
    3. The `outque`, which will holds serialized (*pickled*) return values of each `task`.
2. Creates a pool of "worker" `Processes`, which are responsible for:
    1. Removing tasks from the `inqueue`, which are deserialized, and executing the `task`.
    2. Executing each `task`, and sending the results to the `outqueue`, where it is serialized and stored.
3. Creates 3 `Threads` which manage the above 3 `Queues`:
    1. The `_task_handler` which populates the `inqueue` with pickled `task` objects, from the `taskqueue`
    2. The `_worker_handler` which "reuses" workers by re-creating them once their work is done.
    3. The `_result_handler` which "removes" elements off of the `outqueue`, which are deserialized, and returned to your parent process call to `Pool.map()`.

Re-read the above again and note everywhere you read `serialize`, `deserialize` or `pickle`. Objects must be `serialized` to a `str` before being shuttled to each process, and then that process must `deserialize` that `str` to re-create the object. This needs to happen on the return journey of the data also. That's **2 calls to pickle.dumps()* and **2 calls to pickle.loads()** per task!

> **Note:** Time spent serializing & deserializing is the overhead that we pay in exchange for multiprocessed concurrency. **If this takes longer than the execution of the `convert()` function, we are wasting out time!** Raymond Hettinger explains this well in his talk here, and I build on our toy example in this post to investigate further.

A Tricky Task

When all fails, drop in a debugger. Let's find where `_task_handler` Thread appends a `task` onto the `inqueue`, and investigate the size and composition of this `task`. (github link)

```
@staticmethod
def _handle_tasks(taskqueue, put, outqueue, pool, cache):
    thread = threading.current_thread()

    for taskseq, set_length in iter(taskqueue.get, None):
        task = None
        try:
            # iterating taskseq cannot fail
            for task in taskseq:
                if thread._state:
                    util.debug('task handler found thread._state != RUN')                        break
                try:
                    import ipdb; ipdb.set_trace()
                    put(task)
```

Let's print the 3rd element of our `task`, the single-element `tuple` mentioned above as `(x,)`:

```
ipdb> args_tuple = task[3]
ipdb> elem = args_tuple[0]
ipdb> func = elem[0]
ipdb> func_args = elem[1]
ipdb> func
<bound method IntToBitarrayConverter.convert of <caches.IntToBitarrayConverter object at 0x1122d60f0>>
ipdb> func_args
(612, 640, 176, 806, 372, 895, 345, 15, 173, ... 449)
```

In the call to `put(task)` above, our `func_args` and `func` will be serialized. The serialization of `func_args` is trivial: these are `ints`.

However, on further inspection, our `<bound method...of object...>` should raise concern! This is an instance method, meaning it holds the entire `IntToBitarrayConverter` object:

```
ipdb> func.__self__
<caches.IntToBitarrayConverter object at 0x1122d60f0>
```

Further, there is a **40 MB** `dict` on this object accessed via the `bitstring_cache` attribute:

```
ipdb> func.__self__.bitstring_cache
{0: '0', 1: '1', 2: '10', 3: '11', 4: '100', 5: '101'...1048576: '10000000000000000000'}
ipdb> import sys
ipdb> sys.getsizeof(func.__self__.bitstring_cache) / 1024 / 1024
40
```

This explains our 600% slowdown. We are pickling/unpickling a **40 MB** `dict` 4 times per `task`!

Serial (Performance) Killers

Here is the actual profiled breakdown, produced with blood, sweat and several debug statements in a *non-optimized* local copy of the `multiprocessing` package.

Process	dumps() calls	dumps() time(s)	avg	loads() calls	loads() time(s)	avg
Parent	42	**3m36s**	5.14s	33	4.34s	.13s
1	4	5.37s	1.34s	5	2.98s	.59s
2	4	5.12s	1.28s	5	3.01s	.60s
3	4	5.18s	1.29s	5	3.75s	.75s
4	4	5.74s	1.43s	5	2.95s	.59s
5	4	5.09s	1.27s	5	3.01s	.60s
6	4	5.14s	1.28s	5	2.96s	.59s
7	4	5.79s	1.44s	4	2.96s	.74s
8	4	5.13s	1.28s	5	3.52s	.70s

How exactly did this happen?

OOP: Object-oriented ~~Programming~~ Problems

An **instance method** is a method which is called on an object, and has the object available within the scope of the method. In **C++** and **Javascript**, we access the `bound object` via the variable `this`. In Python, we use `self`.

Recall the method `convert(...)`, on our toy class, found below:

```
def convert(self, integer: int) -> np.ndarray:
    bitstring = self.bitstring_cache[integer]
    return self._bitstring_to_ndarray(bitstring)
```

Once our object `IntToBitarrayConverter` is created, the object is bound to the method `convert(...)`. This means when we pass our method to `Pool.map(...)`, we are implicitly passing *a reference* to the object as well.

Passing an **instance method** to `Pool.map(...)` is **OK**, so long as the **instance** is not large. That said, a rule-of-thumb is to use `@staticmethods` or regular unbound functions when using `Pool`, to explicitly avoid this scenario. Otherwise you are at the mercy of the size of data that consumers add to your object.

The big takeaway here is that we spend **3m 35s** in the *parent process* continuously pickling our `bitstring_cache` so it can be sent to our children.

A Global Solution

The proposed solution of using unbound methods may not be appealing. Often times we need additional state to apply our mapped function `f(x)` across our `iterable`. Each worker process may need to do things like:

1. Acquire a database connection to fetch or update data.
2. Access an in-memory cache, like our `dict` cache example above.

Fortunately, the general concept of **initializaing** each Pool `worker` process so that it has access to an *unpickable* object like a **database engine**, or a *large object* like our **bitstring_cache** is supported, with limitations…

We continue our example here for a solution to our problem using global variables.

A ~~Global~~ Correct Solution

Unfortunately, **global variables** shatter encapsulation, and in my opinion, are not a satisfying solution. I've begun work on a Pull Request augmenting the current `initializer` kwarg used to initialize Pool workers, while maintaining encapsulation. Please check out my work so far, and leave feedback. New test coverage is in progress. I will update when complete once a PR has been opened. No existing tests/interfaces have been broken.