Concurrency and Parallelism

Intermediate Application Development

Otago Polytechnic
Dunedin, New Zealand
Kaiako: Tom Clark

Introduction

Two things that can slow a process down

- Waiting for I/O (I/O bound)
- Heavy computational load (CPU bound)

Concurrency and Parallelism

We can improve performance in some of these cases using concurrency and/or parallelism.

- If our process is waiting on I/O, then we can use concurrency to do something else while waiting
- Sometimes a complex computation can be divided into multiple parts that be run in parallel.
- These techniques can yield great performance gains, but they are somewhat complex and can lead to difficult bugs.

Example: Slow Code

This code, unsurprisingly, spends most of its time waiting.

```
from time import sleep

def slow(x):
    sleep(10)
    return x

def do_tasks(num):
    for n in range(num):
        slow(n)

do_tasks(5)
```

This code is very slow since we call `slow()`, wait for it to finish, and then call it again repeatedly.

Basic Threading

```
import threading
from time import sleep

def slow(x):
    sleep(10)
    return x

def do_threaded_tasks(num):
    threads = []
    for n in range(num):
        t = threading.Thread(target=slow, args=(n,))
        threads.append(t)
        t.start()
    # this next bit is optional
    for t in threads:
        t.join()

do_threaded_tasks(5)
```

Thread Pools

```
from time import sleep
from concurrent.futures import ThreadPoolExecutor

    def slow(x):
        sleep(10)
        return x

    def do_threaded_tasks(num):
        tasks = list(range(num))
        results = None
        with ThreadPoolExecutor(max_workers=10) as ex:
            results = ex.map(slow, tasks)
        return results

    do_threaded_tasks(5)
```

One thing this code does that the previous example did not it collect the return values of the calls to `slow()`.

Race Conditions

```
def slow(x, results):
    sleep(10)
    results.append(x)

def do_threaded_tasks(num):
    threads = []
    results = []
    for n in range(num):
        t = threading.Thread(target=slow, args=(n,results))
        ...

do_threaded_tasks(5)
```

Now we can get at the results, but there may a problem. All the running threads write to the shared results list in an uncontrolled way. This can lead to *race conditions*. However, it turns out the Python lists are *thread safe*.

Locking

When threads work with shared memory we need a way to control access. One basic way to to that is locking.
Suppose that lists were not thread safe.

```
import threading

class ResultList:
    def __init__(self):
        self.results = []
        self._lock = threading.Lock()

    def append(self, result):
        with self._lock:
            self.results.append(result)
```

Python Threads

We tend to think of threads as running concurrently. In some languages that's true, and so threads can speed up code that is I/O bound or CPU bound. In standard Python implementations, however, threads are not truly concurrent. This means that they're really only useful for speeding up I/O bound code.

Process Pools

```
from time import sleep
from multiprocessing import Pool

    def slow(x):
        sleep(10)
        return x

    def do_multiprocess_tasks(num):
        tasks = list(range(num))
        results = None
        with Pool() as p:
            results = p.map(slow, tasks)
        return results

    do_multiprocess_tasks(5)
```

Multiprocessing

- Processes can run on other cores and thus run concurrently
- Since each task is run in a separate process with its own memory, race conditions are less of an issue, but it also means that it is harder to share information.
- The number of processes that can run at one time is limited to the number of cores on the host

References

- Threading:
  https://docs.python.org/3/library/threading.html
- Multiprocessing:
  https://docs.python.org/3/library/multiprocessing.html
- concurrent.futures: https:
  //docs.python.org/3/library/concurrent.futures.html

Programming Activity

1. Pull the course materials repo.
2. Create a new branch, `21-practical` in your practicals repo.
3. Add a subdirectory, `21-practical` and copy `21-practical.ipynb` from the class materials into it.
4. Open a shell, cd to this directory, and run `jupyter notebook` to open the notebook. Complete the first two questions.