Database Systems Lab
Concurrency

Christian Rauch

August 27, 2024

Parallelization

Parallelize computation-heavy portions of your algorithm.

Sometimes you need to spawn and join threads directly.
But preferably use high-level worker/pool APIs:
Pool or ProcessPoolExecutor.

```
1 import threading
2 t = threading.Thread(target=...)
3 t.start() ... t.join()
4
5 import multiprocessing
6 p = multiprocessing.Process(target=...)
7 p.start() ... p.join()
```

Use locks to synchronize resource access and manage the order of acquisition to avoid deadlocks. Use reader/writer locks (i.e., shared/exclusive locks) if possible.

Multi-Processing and Multi-Threading

The Global Interpreter Lock (GIL) allows only one thread to execute bytecode at a time per process. You would therefore often prefer multi-processing over multi-threading in Python.

```
1 from multiprocessing.pool import Pool
2 # alternative: from concurrent.futures import
    ProcessPoolExecutor
```

Share objects between processes/threads and create locks.

```
1 from multiprocessing import Manager
```

Creates new functions from an existing functions with pre-filled arguments (e.g., a lock shared between processes).

```
1 from functools import partial
```

Pools and Locks

```
from flask import g
class StatisticsService():
  def __init__(self): self.da = g.da_factory()
  def generate_chart(self, lock, seller_id): ...
  def generate_seller_statistics(self):
    seller_ids = self.da.read('User',
      lambda u: 'seller' in u.roles, # select
      lambda u: u.id)                # project
```

You may use resources synchronized through mutual exclusion.

```
    manager = Manager()
    lock = manager.Lock()
    gen_chart_lk = partial(self.gen_chart, lock)
```

The pool offers blocking (i.e., map, apply) and asynchronous (i.e., map_async, apply_async) methods for data-parallel execution.

```
    with Pool(8) as pool:
      pool.map(gen_chart_lk, seller_ids)
```

Asynchrony

You may want to free up the current thread by running I/O-bound or otherwise blocking tasks in the background using asyncio futures.

```
1  import asyncio
2
3  async def generate_chart(data):
4    await asyncio.sleep(10) # simulate computation
5    ...
6    return result
7
8  async def generate_product_statistics():
9    pids = ...
10   tasks = [generate_chart(i) for i in pids]
11   results = await asyncio.gather(*tasks)
12   ...
```

Conclusion

Thank you for your attention!