Enhance your Python Code to go beyond GIL

Nitin Bhojwani
Arabinda Das
Priya Pandian
What is Global Interpreter Lock?

The mechanism used by the CPython interpreter to assure that only one thread executes Python bytecode at a time.
More about GIL

• Lock at Interpreter Level
• Prevents true parallelism
• Few Exceptions
  • Extensions Modules
• No GIL in I/O
• Past Efforts
Positive Side of GIL

• Thread Safe
• Single-threaded Programs
• Easy integration of C libraries
• Simplified Garbage Collection
Automation Workloads

Percentage

- I/O: 75%
- Memory: 15%
- CPU: 5%
- Other: 5%

I/O
Memory
CPU
Other
How to Solve

• Multithreading
• Multiprocessing
• AsyncIO
• AsyncIO with Multithreading
• AsyncIO with Multiprocessing
Multithreading

- Multiple child threads
- Shared Memory
- But threads waits for GIL

Advantages
- Lesser memory
- Good for Blocking I/O
Multithreading (how)

• Python’s *threading* module.
• *concurrent.futures.ThreadPoolExecutor* – abstracts queuing and distributing tasks to threads.

```python
eexecutors = concurrent.futures.ThreadPoolExecutor(max_workers=10)

jobs = [executors.submit(call, url) for url in urls]

for job in concurrent.futures.as_completed(jobs):
    # do something with job.result()
```
Multiprocessing

- Multiple child processes
- Message Passing
- Might require more Memory compared to multi-threading.

Advantages
- No GIL
- Good for CPU bound
Multiprocessing (how)

- Python’s `multiprocessing` module

```python
eexecutors = concurrent.futures.ProcessPoolExecutor(max_workers=10)

jobs = [executors.submit(call, url) for url in urls]

for job in concurrent.futures.as_completed(jobs):
    # do something with job.result()
```
Caveats of multithreading & multiprocessing

• Context Switches
  • No. of Threads/Processes -> No. of Context Switches

• Deciding Optimal number of Threads or Processes
  • Varying I/O wait-times
Python AsyncIO
Python AsyncI/O

```
import asyncio
from aiohttp import ClientSession


async def fetch(url):
    async with ClientSession() as session:
        async with session.get(url) as response:
            response = await response.read()
            # do something with response

loop = asyncio.get_event_loop()

tasks = [asyncio.ensure_future(fetch(url)) for url in urls]

loop.run_until_complete(asyncio.wait(tasks))
```

- Python’s `asyncio` module
- `aiohttp` - Asynchronous HTTP Client/Server for asyncio and Python

Advantages
- No wait (Event Driven)
- Good for non-blocking
AsyncI/O with Multithreading/Multiprocessing

```python
# Blocking I/O - Run eventloops in a thread-pool:
with concurrent.futures.ThreadPoolExecutor() as pool:
    result = await loop.run_in_executor(pool, blocking_io)
print('custom thread pool', result)

# CPU Bound - Run eventloops in a process pool:
with concurrent.futures.ProcessPoolExecutor() as pool:
    result = await loop.run_in_executor(pool, cpu_bound)
print('custom process pool', result)
```
Summary

• Multithreading – Blocking I/O
• Multiprocessing – CPU Bound
• AsyncIO – Non-blocking I/O
• AsyncIO with Multithreading – Blocking and Non Blocking I/O
• AsyncIO with Multiprocessing – CPU Bound with Non Blocking I/O
Reach out to Us

• Nitin Bhojwani – nitinbhojwani@outlook.com
• Arabinda Das – darabinda@gmail.com
Thank You!

Q & A