facebook

Thinking Outside the GIL

With AsyncIO and Multiprocessing

John Reese

Production Engineer, Facebook

@n7cmdr

github.com/jreese

What’s the GIL?

- Global Interpreter Lock
- One VM thread at a time
- No concurrent memory access
- I/O wait releases lock

Stateful monitoring

- Gather ~100M data points
- Process and aggregate anomalies
- Easy to add new checks
- Simple deployment
- Few dependencies

```python
def fetch(url):
    return requests.get(url)

def process(url, response):
    if 200 ≤ response.status ≤ 299:
        ...  # return anomaly

# store anomalies in sql somewhere
```

```python
def fetch(url):
    return requests.get(url)

def process(url, response):
    if 200 ≤ response.status ≤ 299:
        ...  # return anomaly

# store anomalies in sql somewhere
```

```python
def fetch(url):
    return requests.get(url)

def process(url, response):
    if 200 ≤ response.status ≤ 299:
        ...  # return anomaly

# store anomalies in sql somewhere
```

#impact

- One binary
- Fetch the world
- Process everything
- Aggregate results
- Thread pool for I/O

Not aging well

- Scales in time and memory
- Runtime now too slow
- Underutilizing hardware
- Ultimately limited by the GIL

Give me options

Switch to py3

~45% memory savings

~20% runtime reduction

Sharding

- Technically correct
- Scales with number of workers
- Complicated deployments
- Communication overhead

Multiprocessing

- Scales with CPU cores
- Automatic IPC
- Pool.map is really useful

```python
def fetch(url):
    return requests.get(url)

def fetch_all(urls):
    with multiprocessing.Pool() as pool:
        results = pool.map(fetch, urls)
```

```python
def fetch(url):
    return requests.get(url)

def fetch_all(urls):
    with multiprocessing.Pool() as pool:
        results = pool.map(fetch, urls)
```

Multiprocessing

- Scales with CPU cores
- Automatic IPC
- Pool.map is really useful
- One task per process
- Beware forking, pickling

AsyncIO

- Based on futures
- Faster than threads
- Massive I/O concurrency

```python
async def fetch_url(url):
    return await aiohttp.request("GET", url)


async def fetch_two(url_a, url_b):
    future_a = fetch_url(url_a)
    future_b = fetch_url(url_b)
    a, b = await asyncio.gather(future_a, future_b)
    return a, b
```

```python
async def fetch_url(url):
    return await aiohttp.request("GET", url)

async def fetch_two(url_a, url_b):
    future_a = fetch_url(url_a)
    future_b = fetch_url(url_b)
    a, b = await asyncio.gather(future_a, future_b)
    return a, b
```

```python
async def fetch_url(url):
    return await aiohttp.request("GET", url)

async def fetch_two(url_a, url_b):
    future_a = fetch_url(url_a)
    future_b = fetch_url(url_b)
    a, b = await asyncio.gather(future_a, future_b)
    return a, b
```

```python
async def fetch_url(url):
    return await aiohttp.request("GET", url)

async def fetch_two(url_a, url_b):
    future_a = fetch_url(url_a)
    future_b = fetch_url(url_b)
    a, b = await asyncio.gather(future_a, future_b)
    return a, b
```

AsyncIO

- Based on futures
- Faster than threads
- Massive I/O concurrency
- Processing still limited by GIL
- Beware timeouts and queue length

Why not both?

Multiprocessing + AsyncIO

- Use multiprocessing primitives
- Event loop per process
- Queues for work/results
- Highly parallel workload
- Need to do some plumbing

```python
async def run_loop(tx, rx):
    ...  # real work here

def bootstrap(tx, rx):
    loop = asyncio.new_event_loop()
    asyncio.set_event_loop(loop)
    loop.run_until_complete(run_loop(tx, rx))

def main():
    p = multiprocessing.Process(
        target=bootstrap,
        args=(tx, rx)
    )
    p.start()
```

```python
async def run_loop(tx, rx):
    ...  # real work here

def bootstrap(tx, rx):
    loop = asyncio.new_event_loop()
    asyncio.set_event_loop(loop)
    loop.run_until_complete(run_loop(tx, rx))

def main():
    p = multiprocessing.Process(
        target=bootstrap,
        args=(tx, rx)
    )
    p.start()
```

```python
async def run_loop(tx, rx):
    ...  # real work here

def bootstrap(tx, rx):
    loop = asyncio.new_event_loop()
    asyncio.set_event_loop(loop)
    loop.run_until_complete(run_loop(tx, rx))

def main():
    p = multiprocessing.Process(
        target=bootstrap,
        args=(tx, rx)
    )
    p.start()
```

```python
async def run_loop(tx, rx):
    limit = 10
    pending = set()
    while True:
        while len(pending) < limit:
            task = tx.get_nowait()
            fn, args, kwargs = task
            pending.add(fn(*args, **kwargs))

        done, pending = await asyncio.wait(pending, ...)
        for future in done:
            rx.put_nowait(await future)
```

```python
async def run_loop(tx, rx):
    limit = 10
    pending = set()
    while True:
        while len(pending) < limit:
            task = tx.get_nowait()
            fn, args, kwargs = task
            pending.add(fn(*args, **kwargs))

        done, pending = await asyncio.wait(pending, ...)
        for future in done:
            rx.put_nowait(await future)
```

```python
async def run_loop(tx, rx):
    limit = 10
    pending = set()
    while True:
        while len(pending) < limit:
            task = tx.get_nowait()
            fn, args, kwargs = task
            pending.add(fn(*args, **kwargs))

        done, pending = await asyncio.wait(pending, ...)
        for future in done:
            rx.put_nowait(await future)
```

```python
async def run_loop(tx, rx):
    limit = 10
    pending = set()
    while True:
        while len(pending) < limit:
            task = tx.get_nowait()
            fn, args, kwargs = task
            pending.add(fn(*args, **kwargs))

        done, pending = await asyncio.wait(pending, ...)
        for future in done:
            rx.put_nowait(await future)
```

```python
async def run_loop(tx, rx):
    limit = 10
    pending = set()
    while True:
        while len(pending) < limit:
            task = tx.get_nowait()
            fn, args, kwargs = task
            pending.add(fn(*args, **kwargs))

        done, pending = await asyncio.wait(pending, ... )
        for future in done:
            rx.put_nowait(await future)
```

```python
async def run_loop(tx, rx):
    limit = 10
    pending = set()
    while True:
        while len(pending) < limit:
            task = tx.get_nowait()
            fn, args, kwargs = task
            pending.add(fn(*args, **kwargs))

        done, pending = await asyncio.wait(pending, ...)
        for future in done:
            rx.put_nowait(await future)
```

```python
async def fetch_url(url):
    return await aiohttp.request('GET', url)

def fetch_all(urls):
    tx, rx = Queue(), Queue()
    Process(
        target=bootstrap,
        args=(tx, rx),
    ).start()

    for url in urls:
        task = fetch_url, (url,), {}
        tx.put_nowait(task)

    ...  # consume response queue
```

```python
async def fetch_url(url):
    return await aiohttp.request('GET', url)

def fetch_all(urls):
    tx, rx = Queue(), Queue()
    Process(
        target=bootstrap,
        args=(tx, rx),
    ).start()

    for url in urls:
        task = fetch_url, (url,), {}
        tx.put_nowait(task)

    ...  # consume response queue
```

```python
async def fetch_url(url):
    return await aiohttp.request('GET', url)

def fetch_all(urls):
    tx, rx = Queue(), Queue()
    Process(
        target=bootstrap,
        args=(tx, rx),
    ).start()

    for url in urls:
        task = fetch_url, (url,), {}
        tx.put_nowait(task)

    ...  # consume response queue
```

```python
async def fetch_url(url):
    return await aiohttp.request('GET', url)

def fetch_all(urls):
    tx, rx = Queue(), Queue()
    Process(
        target=bootstrap,
        args=(tx, rx),
    ).start()

    for url in urls:
        task = fetch_url, (url,), {}
        tx.put_nowait(task)

    ...  # consume response queue
```

```python
class Pool:
    async def queue(self, fn, *args, **kwargs) → int:  ...
    async result(self, id) → Any:  ...

    async def map(self, fn, items):
        task_ids = [
            await self.queue(fn, (item,), {})
            for item in items
        ]

        return [
            await self.result(task_id)
            for task_id in task_ids
        ]
```

```python
class Pool:
    async def queue(self, fn, *args, **kwargs) -> int: ...
    async def result(self, id) -> Any: ...

    async def map(self, fn, items):
        task_ids = [
            await self.queue(fn, (item,), {})
            for item in items
        ]

        return [
            await self.result(task_id)
            for task_id in task_ids
        ]
```

```python
class Pool:
    async def queue(self, fn, *args, **kwargs) → int:  ...
    async def result(self, id) → Any:  ...

    async def map(self, fn, items):
        task_ids = [
            await self.queue(fn, (item,), {})
            for item in items
        ]

        return [
            await self.result(task_id)
            for task_id in task_ids
        ]
```

```python
class Pool:
    async def queue(self, fn, *args, **kwargs) → int:  ...
    async def result(self, id) → Any:  ...

    async def map(self, fn, items):
        task_ids = [
            await self.queue(fn, (item,), {})
            for item in items
        ]

        return [
            await self.result(task_id)
            for task_id in task_ids
        ]
```

```python
async def fetch_url(url):
    return await aiohttp.request('GET', url)

async def fetch_all(urls):
    async with Pool() as pool:
        results = await pool.map(fetch_url, urls)
```

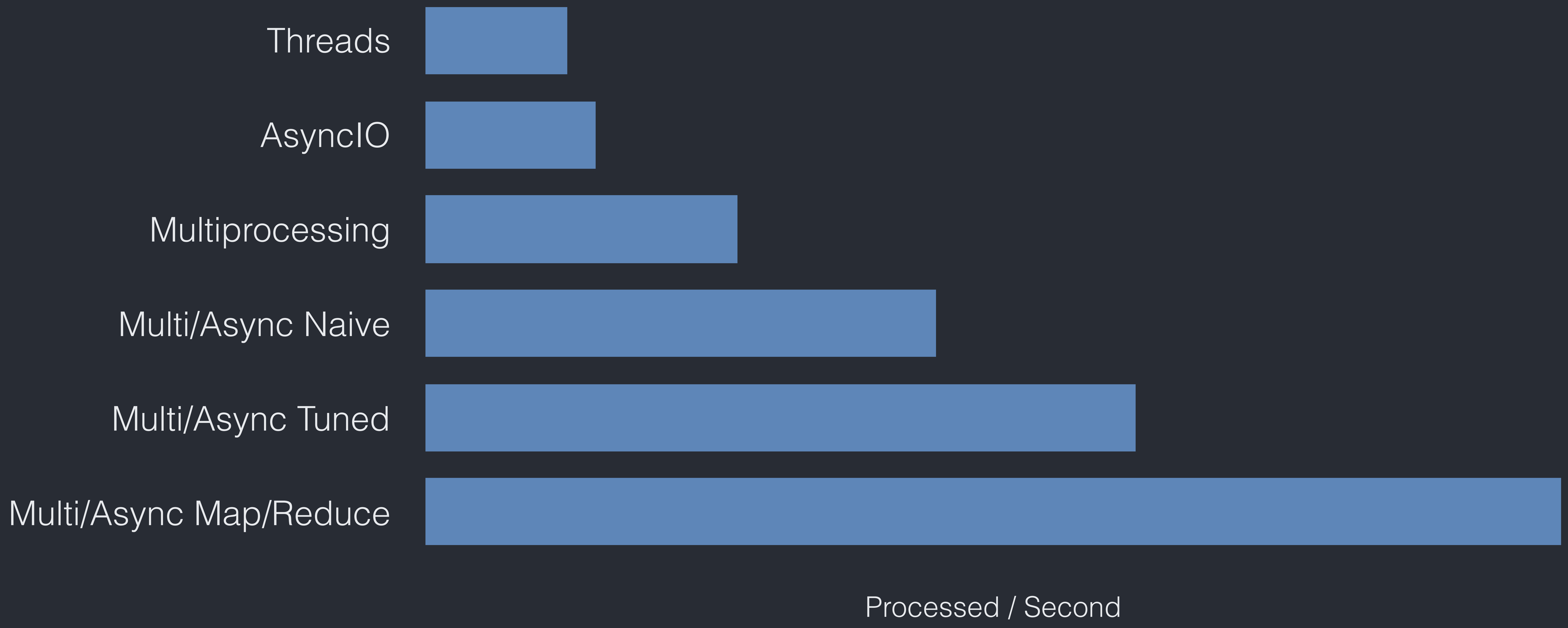
Optimizations

- Multiple work queues
- Combine tasks into batches
- Use spawned processes

Considerations

- Minimize what you pickle
- Prechunk work items
- Aggregate results in the child
- Use map/reduce

Performance comparison



I want it!

```
$ pip install aiomultiprocess
```

aiomultiprocess

github.com/jreese/aiomultiprocess

- Simple implementation
- Emulates multiprocessing API
- One shot or process pool
- Supports map/reduce workloads

```python
from aiomultiprocess import Pool

async def fetch_url(url):
    return await aiohttp.request('GET', url)

async def fetch_all(urls):
    async with Pool() as pool:
        results = await pool.map(fetch_url, urls)
```

Python is slow

Python is ~~slow~~ powerful

Great tools make complex tasks simple

John Reese

Production Engineer, Facebook

@n7cmdr

github.com/jreese

facebook