# The GIL and API Performance

The Past, Present, and Free-Threaded Future

## whoami

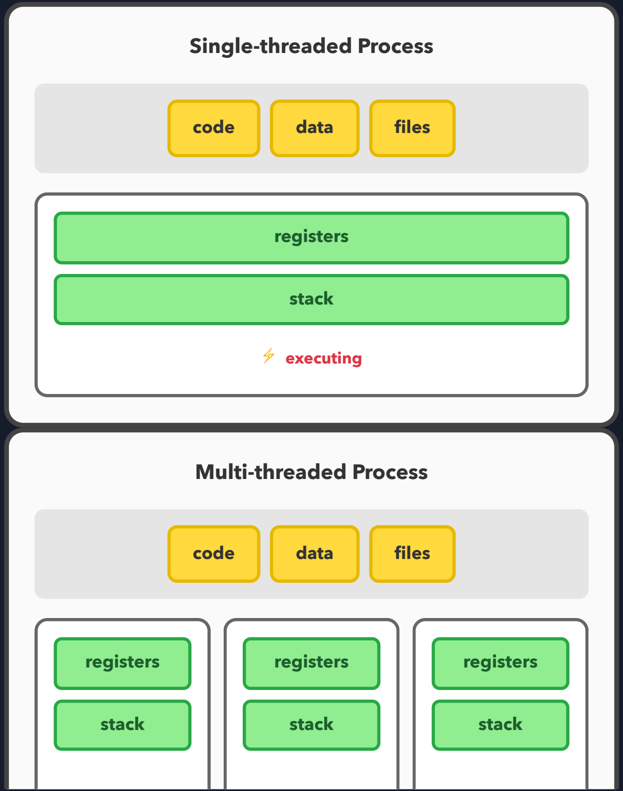
**Ruben Hias**

Software Engineer at TechWolf

**floww.dev**: Workflow automation for developers

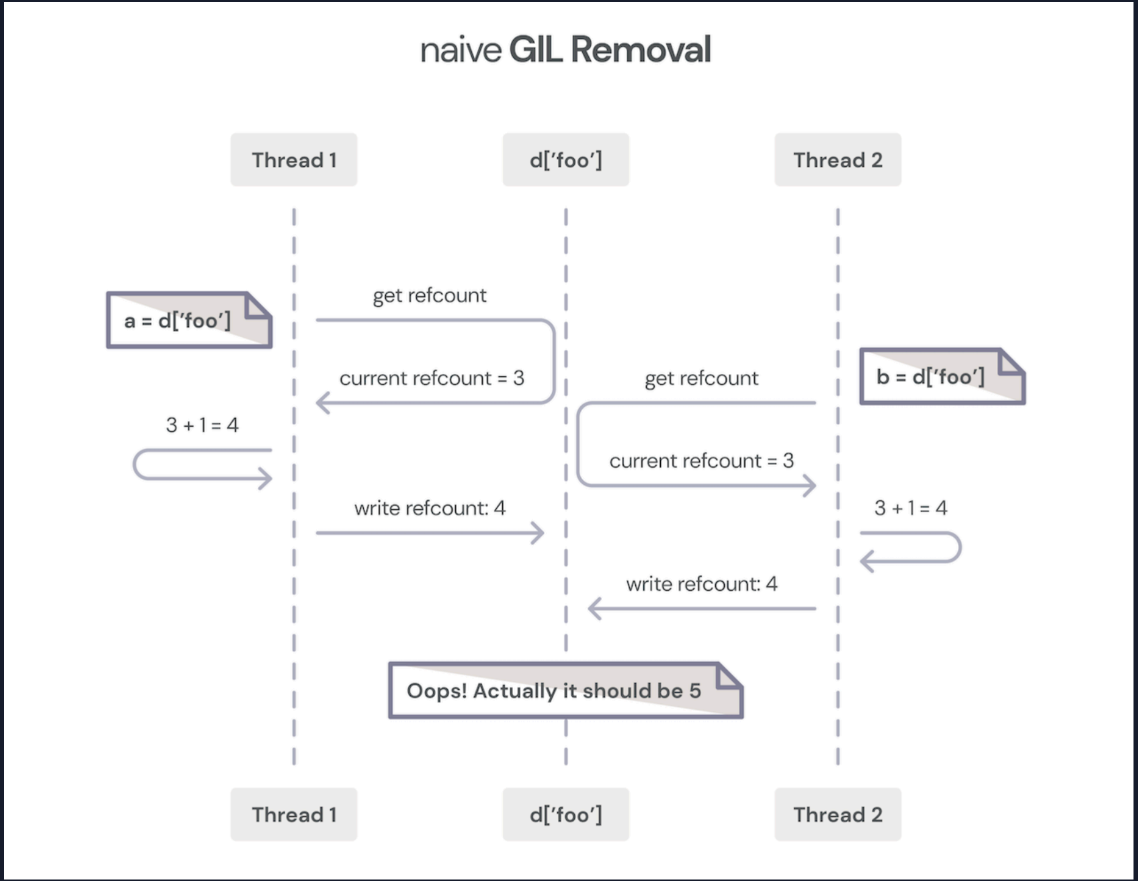
# Refresher: Threads vs Processes

- **Process**: Independent program with its own memory space
  - Isolated memory → safer but higher overhead
  - Communication via IPC
- **Thread**: Lightweight execution unit within a process
  - Shared memory space → efficient but requires synchronization
  - Direct communication via shared data

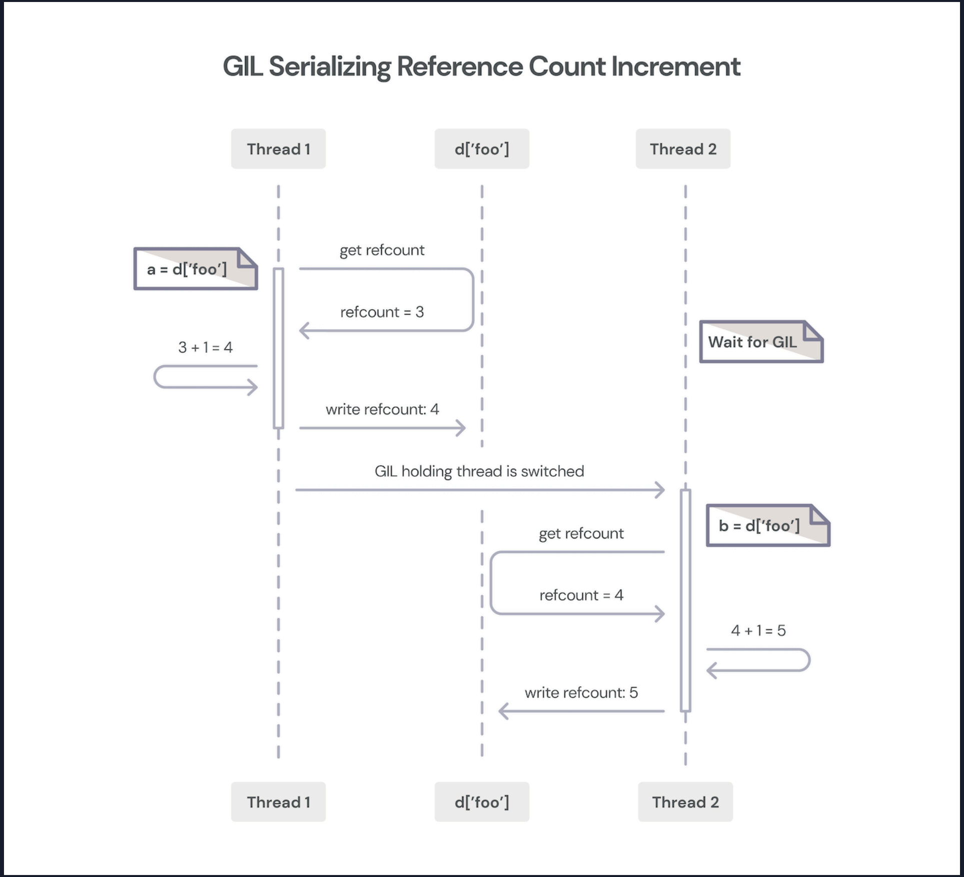


## GIL: Global Interpreter Lock

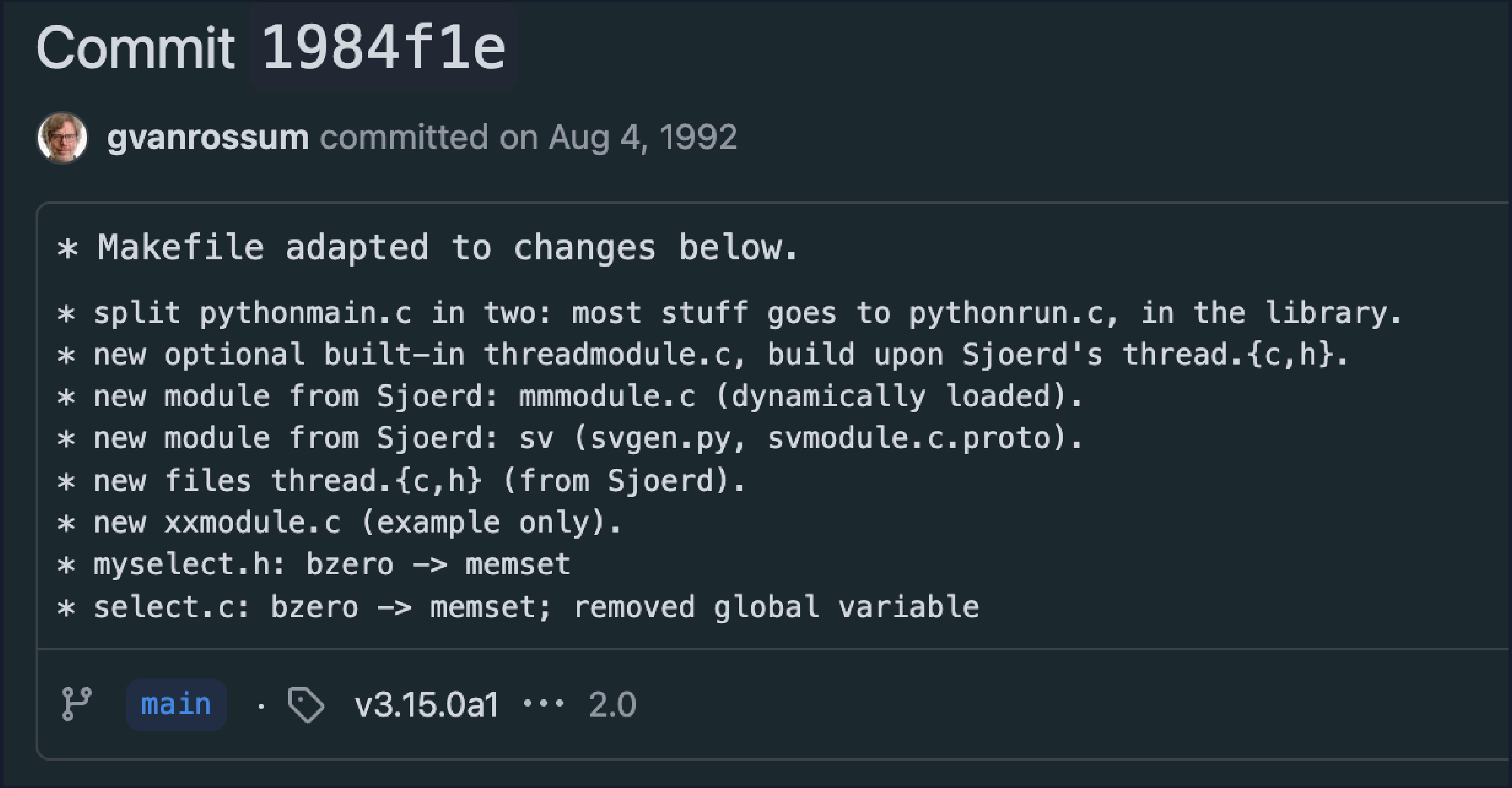
- *A lock that allows only* ***one thread*** *to execute Python bytecode at a time*
- CPython Implementation detail
- Protects internal data structures



[1] The Python GIL - Backblaze



[1] The Python GIL - Backblaze

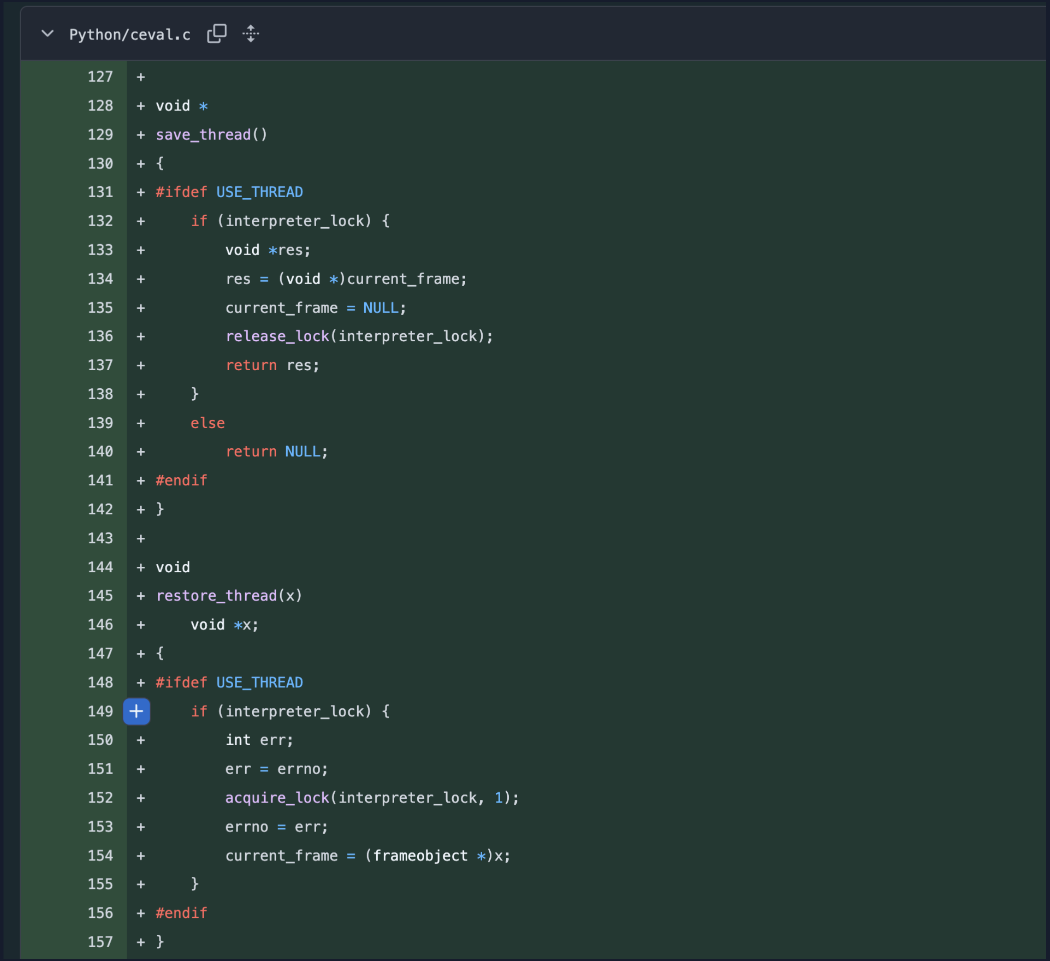

Commit 1984f1e

gvanrossum committed on Aug 4, 1992

```
* Makefile adapted to changes below.

* split pythonmain.c in two: most stuff goes to pythonrun.c, in the library.
* new optional built-in threadmodule.c, build upon Sjoerd's thread.{c,h}.
* new module from Sjoerd: mmmodule.c (dynamically loaded).
* new module from Sjoerd: sv (svgen.py, svmodule.c.proto).
* new files thread.{c,h} (from Sjoerd).
* new xxmodule.c (example only).
* myselect.h: bzero -> memset
* select.c: bzero -> memset; removed global variable
```

main · v3.15.0a1 ··· 2.0


Python/ceval.c

```diff
+
+ void *
+ save_thread()
+ {
+ #ifdef USE_THREAD
+     if (interpreter_lock) {
+         void *res;
+         res = (void *)current_frame;
+         current_frame = NULL;
+         release_lock(interpreter_lock);
+         return res;
+     }
+     else
+         return NULL;
+ #endif
+ }
+
+ void
+ restore_thread(x)
+     void *x;
+ {
+ #ifdef USE_THREAD
+     if (interpreter_lock) {
+         int err;
+         err = errno;
+         acquire_lock(interpreter_lock, 1);
+         errno = err;
+         current_frame = (frameobject *)x;
+     }
+ #endif
+ }
```

## Why even have threads then?

## Why even have threads then?

### 1. I/O

- HDD Read speed: 0.5-2 MB/s
- Network Latency: 100’s of ms
- Network Bandwidth: 14.4 KB/s

## Why even have threads then?

## Why even have threads then?

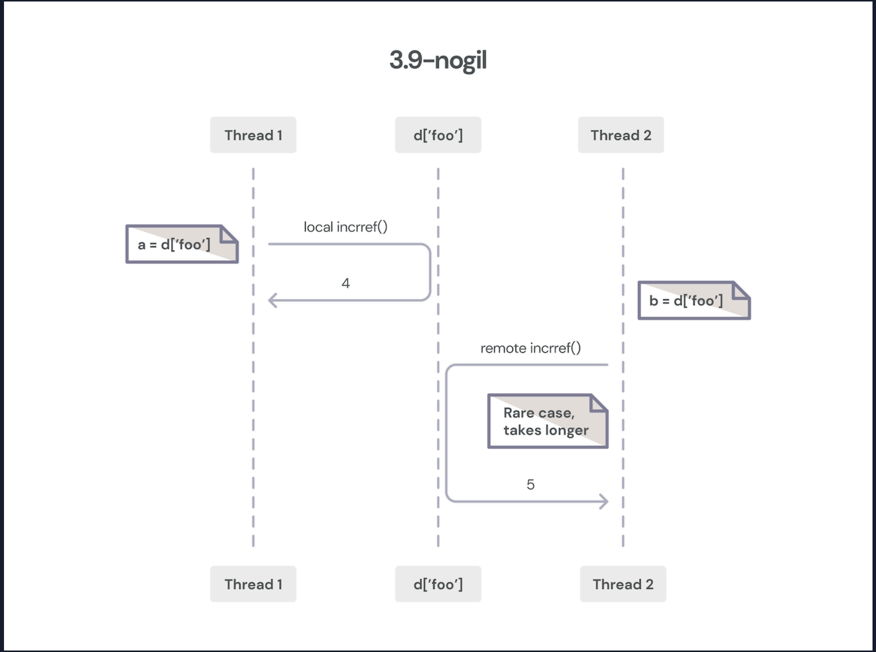
3. Simple to Extend

## GIL removal attempts

- 1996
- 2007
- 2015 (x2)
- 2021

## Core innovations

- `nogil` ~ Sam Gross
- Biased reference counting & immortal objects
- Thread-safe memory allocator
- Many more small tweaks
- **Officially accepted as PEP**



## Recap: How we got here

- GIL is a CPython implementation detail to protect internal state
- Multiple removal attempts failed (1996, 2007, 2015) due to performance penalties
- Python 3.13 finally achieves free-threading via biased reference counting
- Trade-off: small single-threaded slowdown for true parallelism
- Now the question is: How do we use this?

```
uv run -p 3.14t main.py
```

## Basic multi-threading

```python
from concurrent.futures import ThreadPoolExecutor

N_THREADS = 8


def do_work(n: int = 10_000_000) -> int:
    return sum(i * i for i in range(n))


with ThreadPoolExecutor(max_workers=N_THREADS) as ex:
    futures = [
        ex.submit(do_work)
        for _ in range(N_THREADS)
    ]
```

**3.14**: 2.352 s ± 0.014 s

**3.14t**: 768.9 ms ± 11.7 ms

**~3x speedup**

## Single-threaded

```python
def do_work(n: int = 10_000_000) → int:
    return sum(i * i for i in range(n))

for i in range(8):
    do_work()
```

**3.14**: 2.341 s ± 0.022 s

**3.14t**: 2.370 s ± 0.043 s

**1.2% overhead**

## Freethreading for CPU-bound applications

- True parallelism possible: 3x+ speedup for parallel CPU-bound tasks
- Best for embarrassingly parallel workloads (independent calculations)
- Single-threaded overhead now minimal (~3% vs Python 3.14)
- Thread overhead matters: use thread pools, avoid creating too many threads

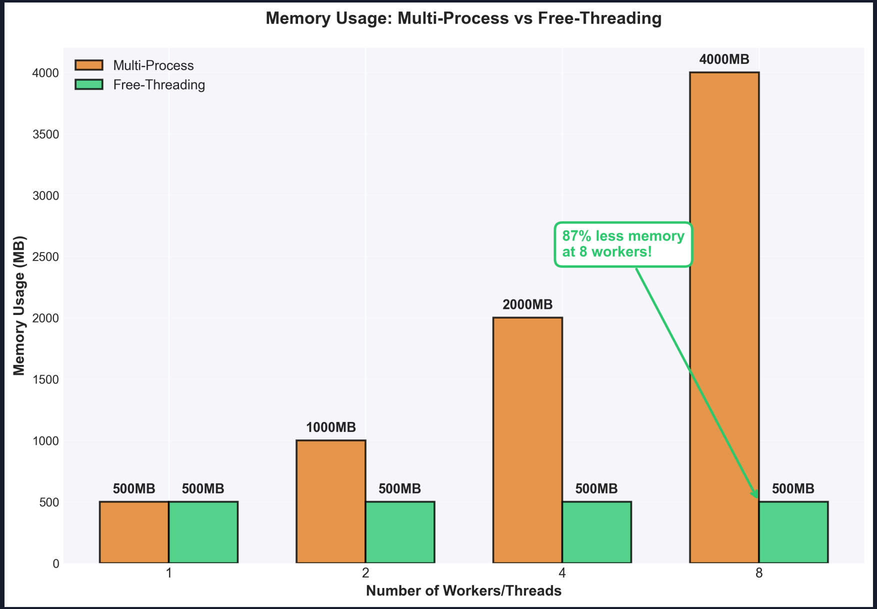
## For the API developers

## How Python APIs Scale Today

- Concurrent requests require parallel execution
- GIL blocks thread-based parallelism
- Solution: spawn multiple worker processes (Gunicorn, uWSGI)
- But processes have trade-offs…

# Issues with processes

- Processes don't share memory space
- Data sharing is hard
- Communication between processes is slow



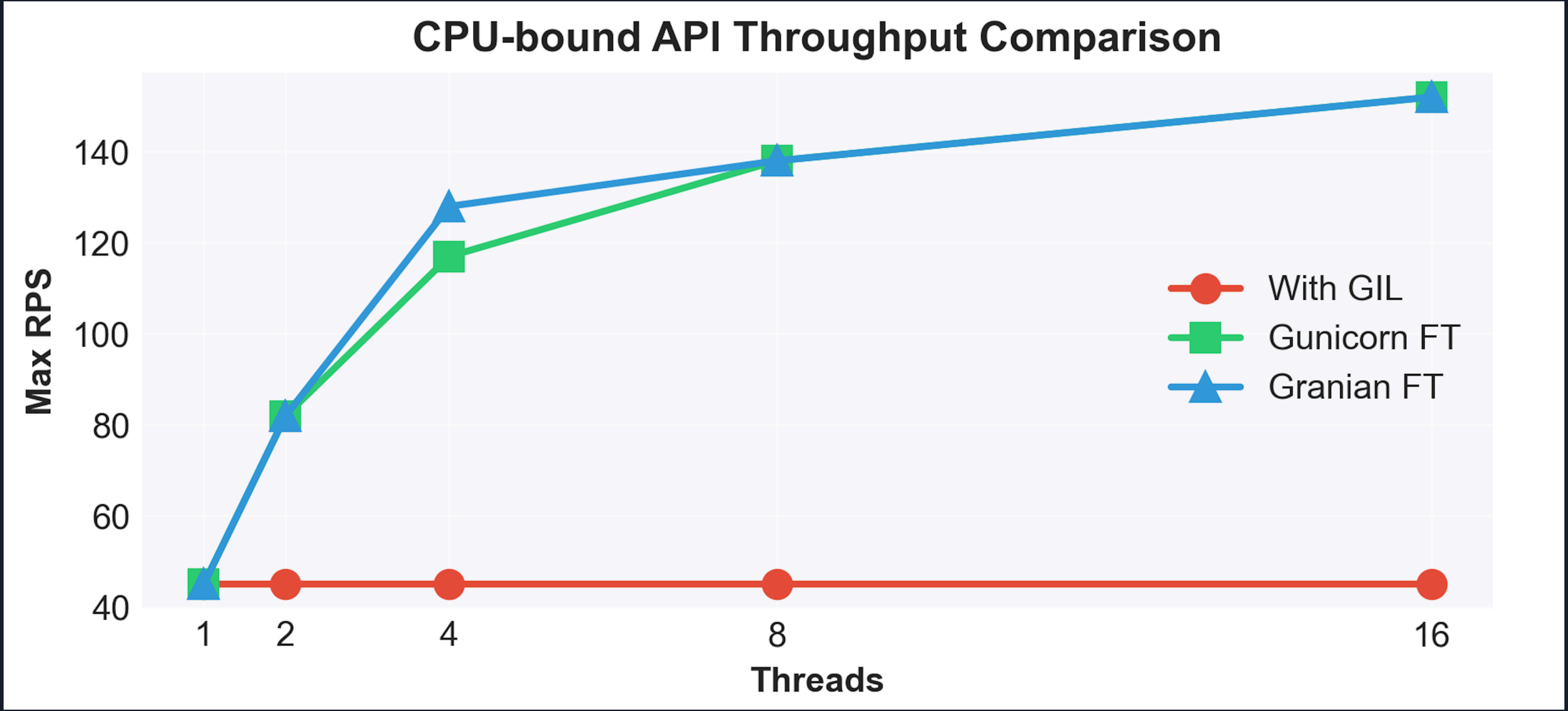
BUT doesn’t have GIL contention

## Benchmark Time

**Configurations tested:**

- Traditional: Gunicorn threads, Granian threads
- Async: Uvicorn/Granian ASGI, Gevent
- Free-threading: Python 3.14t with threads
- Baseline: Multi-process (current best practice)

**Question:** Can free-threading match multi-process performance?

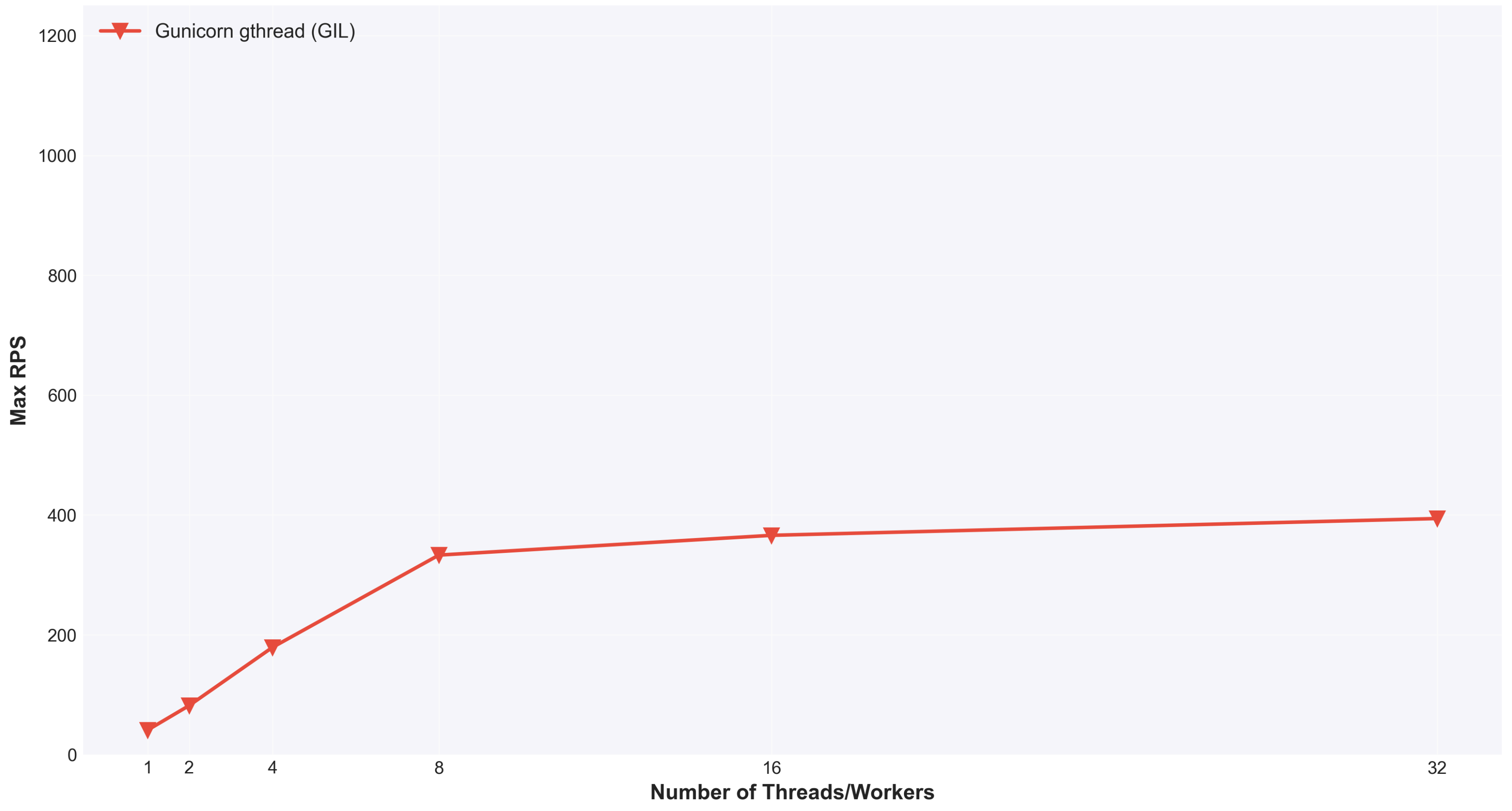

CPU-bound API Throughput Comparison
Max RPS
140
120
100
80
60
40
1
2
4
8
16
Threads
With GIL
Gunicorn FT
Granian FT

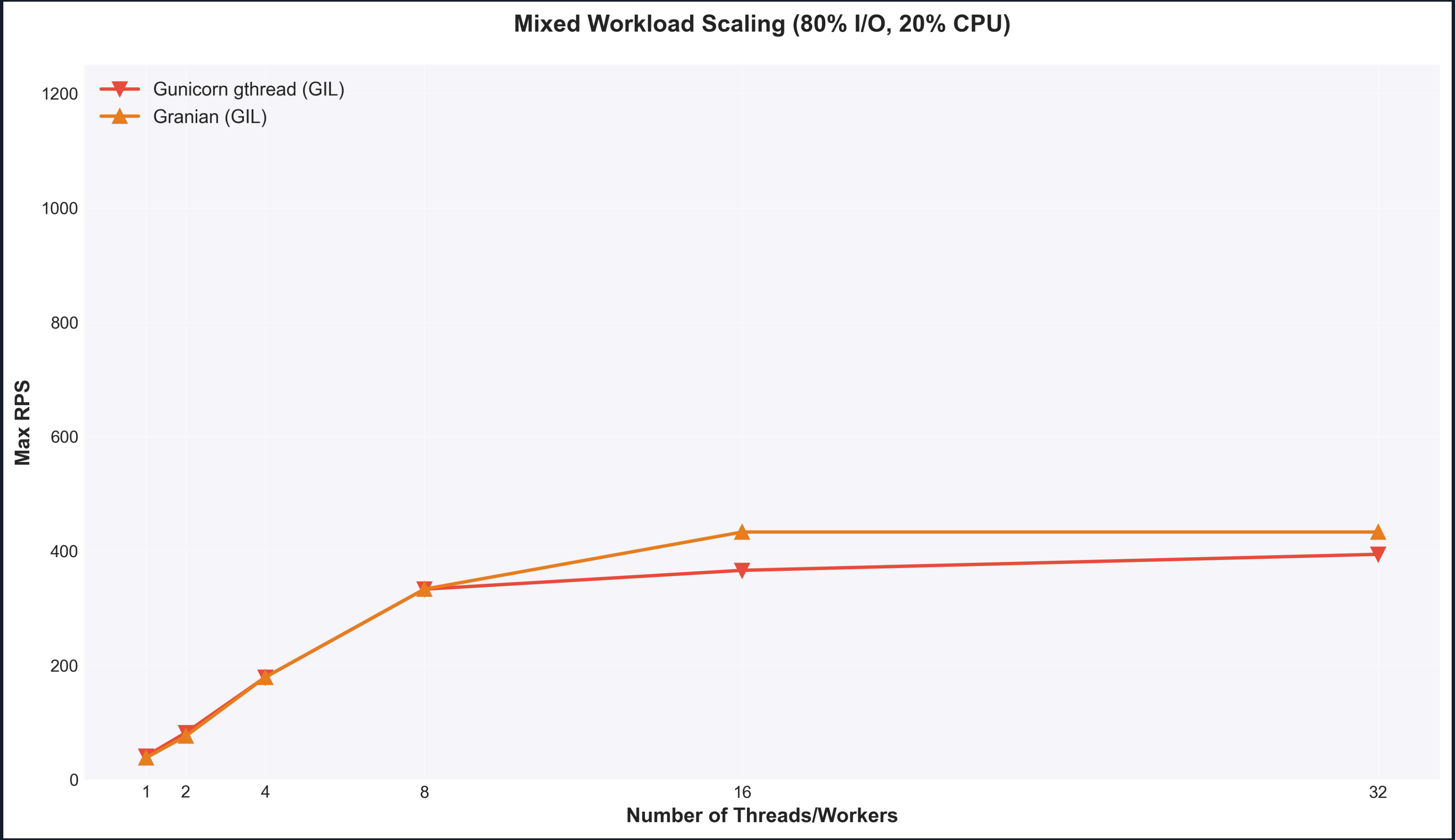
## Benchmark: Mixed Workload

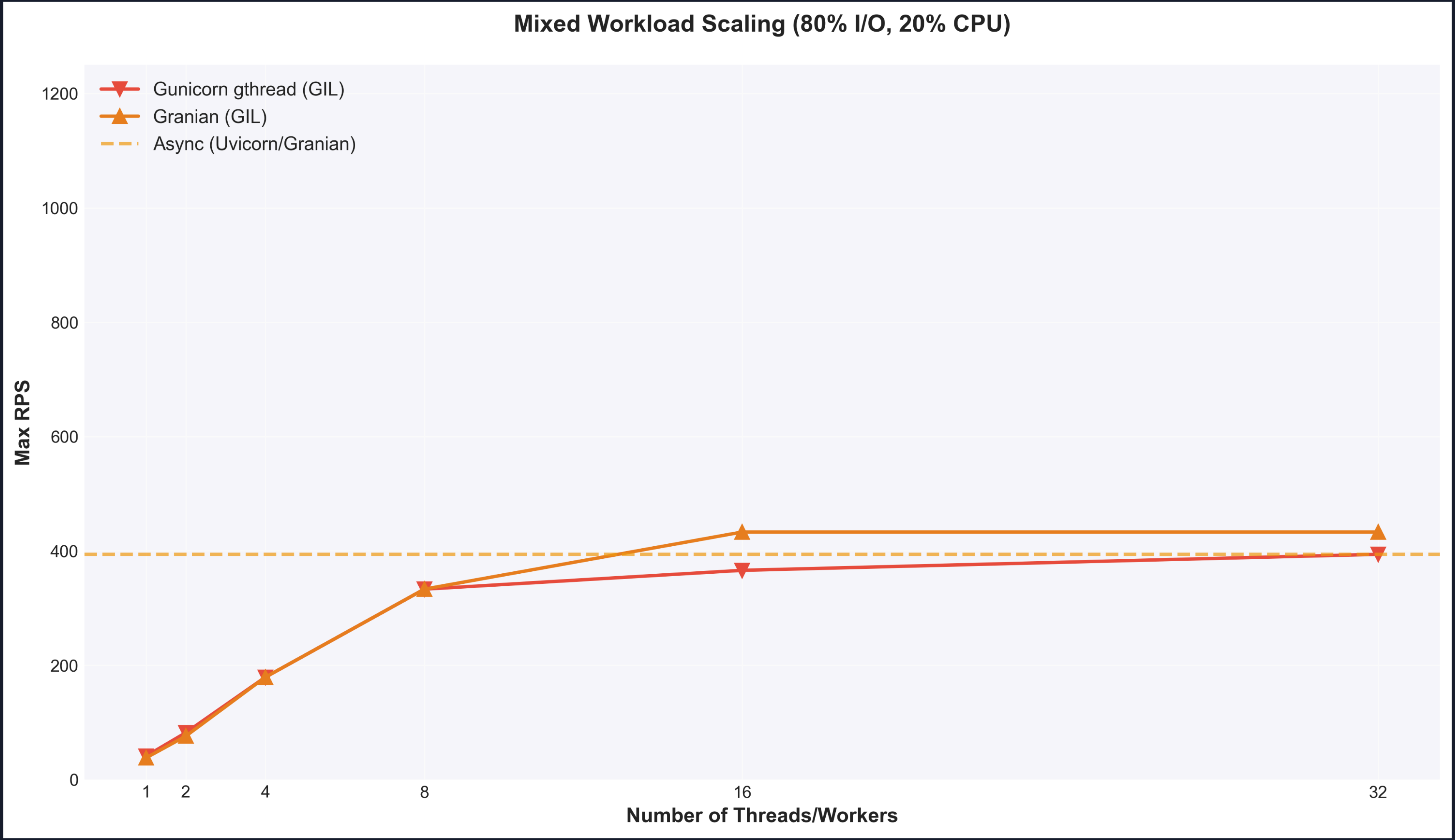
Testing a realistic API: 80% I/O operations, 20% CPU work

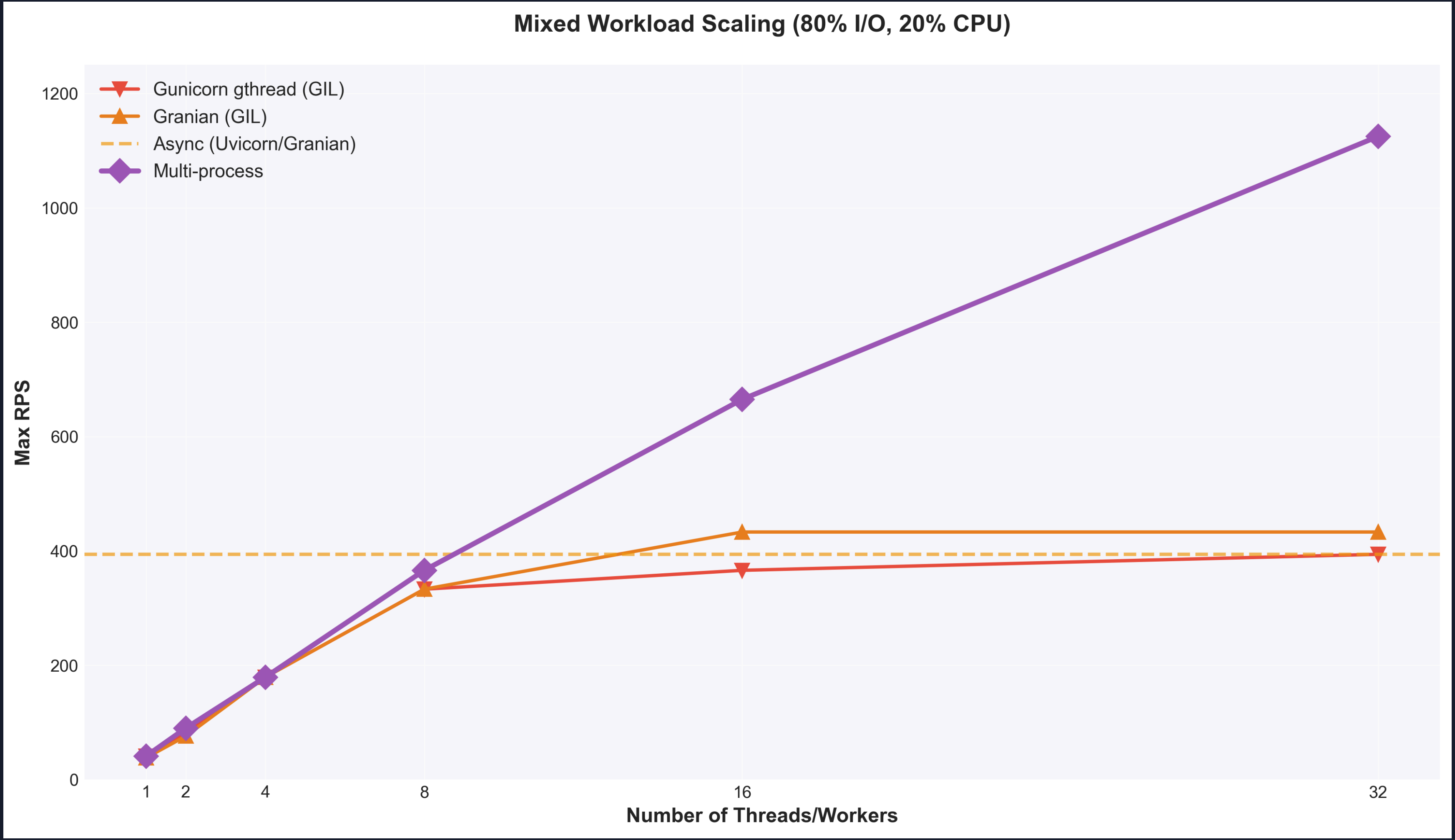
- Comparing different scaling strategies
- Varies thread/worker count from 1 to 32
- Measures maximum requests per second (RPS)

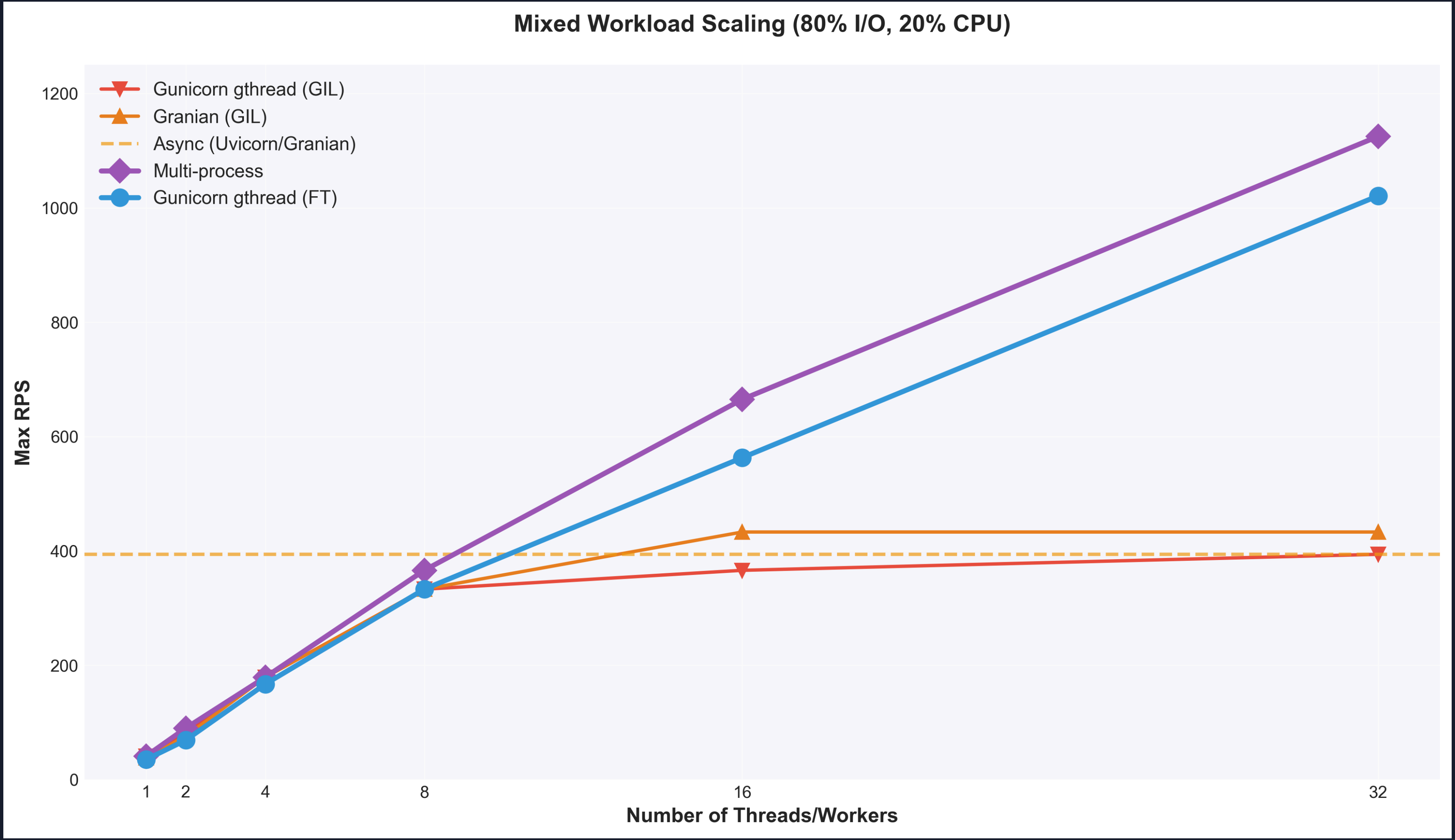
## Mixed Workload Scaling (80% I/O, 20% CPU)

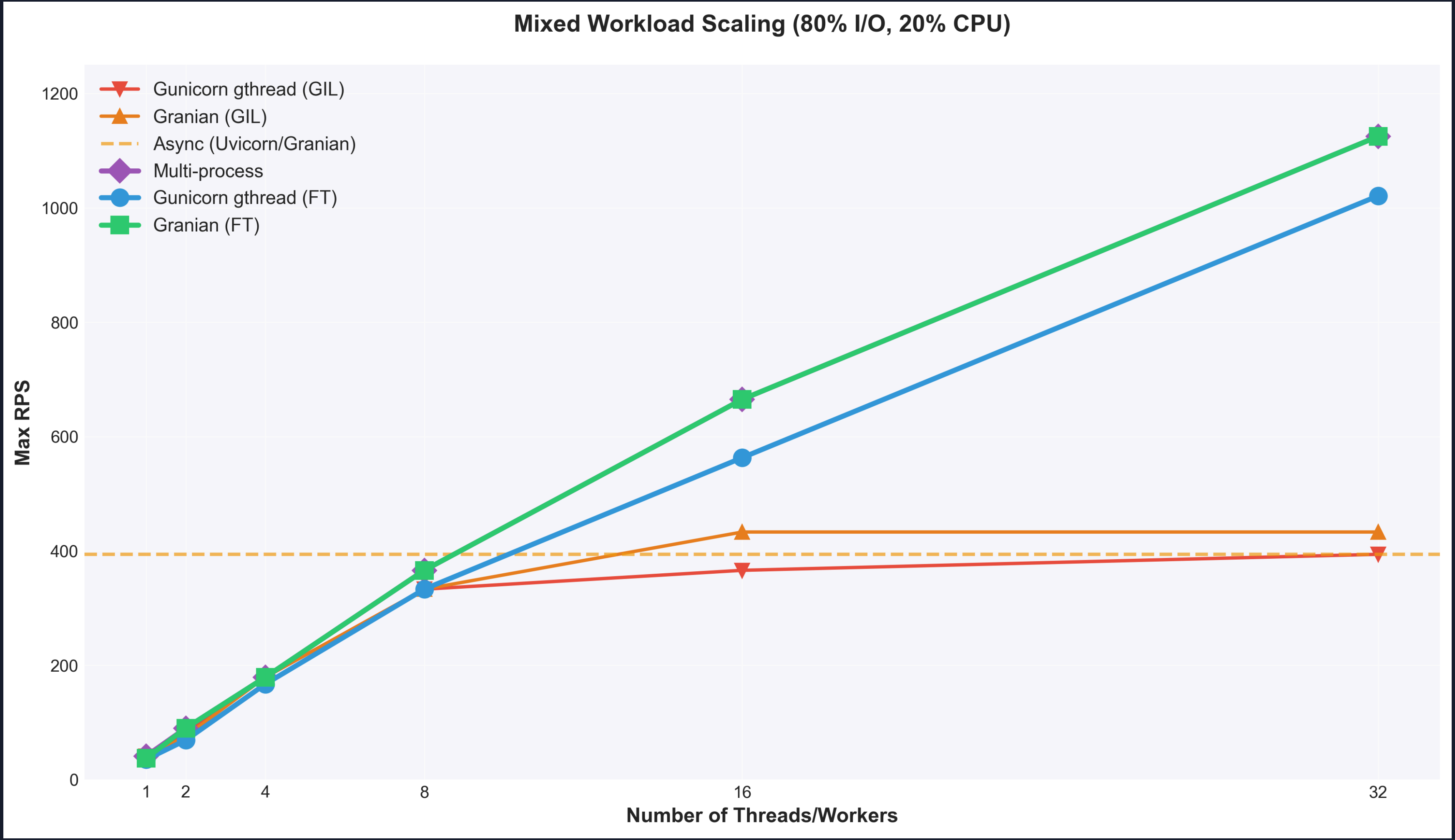



Mixed Workload Scaling (80% I/O, 20% CPU)
Gunicorn gthread (GIL)
Granian (GIL)
Max RPS
0
200
400
600
800
1000
1200
1
2
4
8
16
32
Number of Threads/Workers

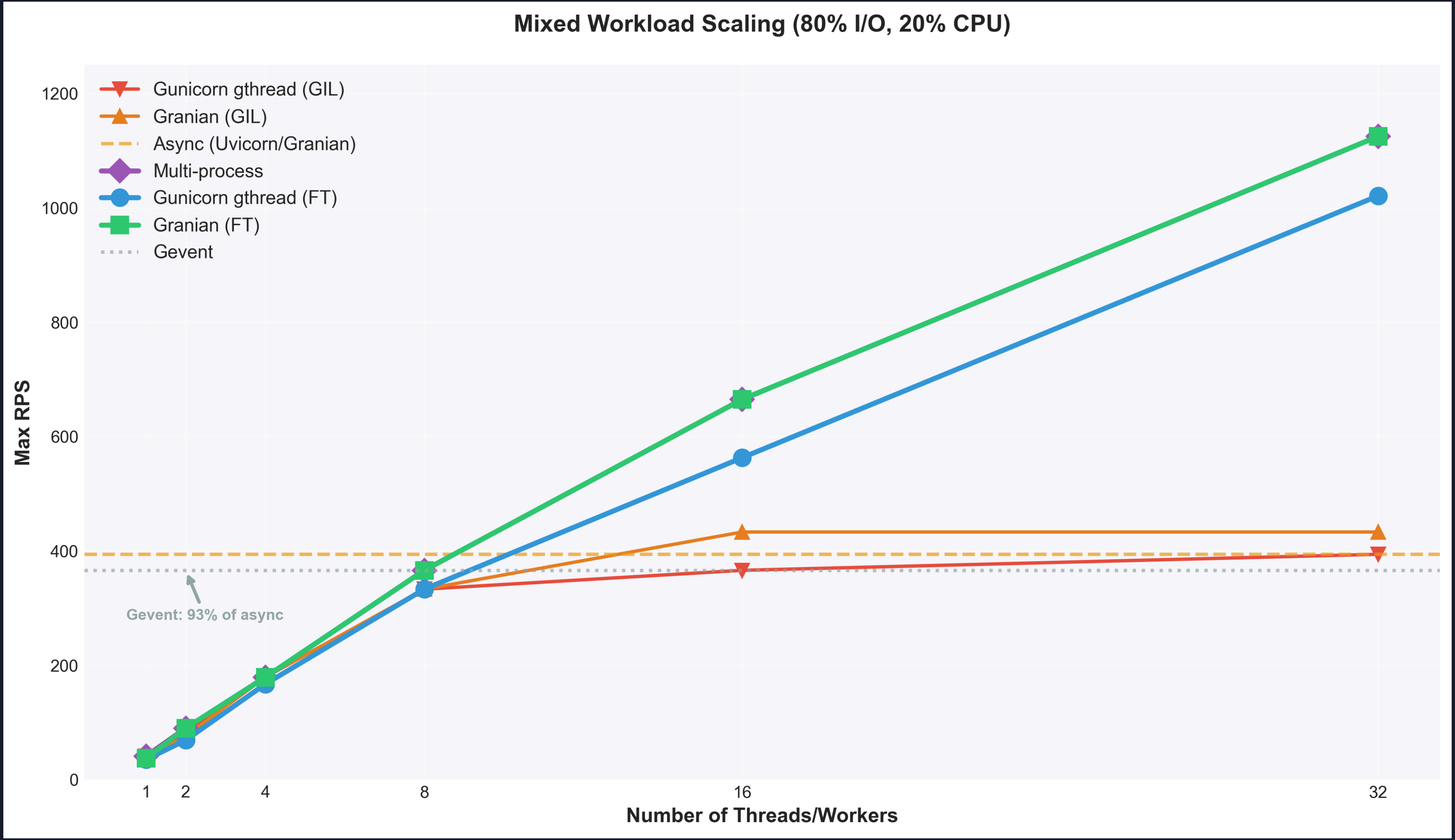

Mixed Workload Scaling (80% I/O, 20% CPU)
Gunicorn gthread (GIL)
Granian (GIL)
Async (Uvicorn/Granian)
Max RPS
0
200
400
600
800
1000
1200
1
2
4
8
16
32
Number of Threads/Workers


Mixed Workload Scaling (80% I/O, 20% CPU)
Gunicorn gthread (GIL)
Granian (GIL)
Async (Uvicorn/Granian)
Multi-process
Max RPS
0
200
400
600
800
1000
1200
Number of Threads/Workers
1
2
4
8
16
32


Mixed Workload Scaling (80% I/O, 20% CPU)
Gunicorn gthread (GIL)
Granian (GIL)
Async (Uvicorn/Granian)
Multi-process
Gunicorn gthread (FT)
Max RPS
0
200
400
600
800
1000
1200
1
2
4
8
16
32
Number of Threads/Workers



Mixed Workload Scaling (80% I/O, 20% CPU)
Gunicorn gthread (GIL)
Granian (GIL)
Async (Uvicorn/Granian)
Multi-process
Gunicorn gthread (FT)
Granian (FT)
Max RPS
0
200
400
600
800
1000
1200
1
2
4
8
16
32
Number of Threads/Workers


Mixed Workload Scaling (80% I/O, 20% CPU)
Gunicorn gthread (GIL)
Granian (GIL)
Async (Uvicorn/Granian)
Multi-process
Gunicorn gthread (FT)
Granian (FT)
Gevent
Gevent: 93% of async
Max RPS
0
200
400
600
800
1000
1200
1
2
4
8
16
32
Number of Threads/Workers

## Key Takeaways

- GIL creates a scaling ceiling regardless of server choice
- Async helps with I/O concurrency but can’t escape CPU-bound GIL limits
- Free-threading enables linear scaling: 2.5x improvement at 32 threads
- For I/O-heavy APIs: async/Gevent remain excellent choices today
- Modern webservers like Granian can better leverage threads

## Conclusions

- The GIL was the right trade-off in 1997, but hardware has changed
- Free-threading delivers true parallelism without memory overhead
- 2.5x scaling improvements for mixed workloads in production-ready Python
- For new projects: consider free-threading. For existing: async/multi-process work well

The future of Python performance is multi-threaded