GIL-free scaling of Uproot in Python 3.13

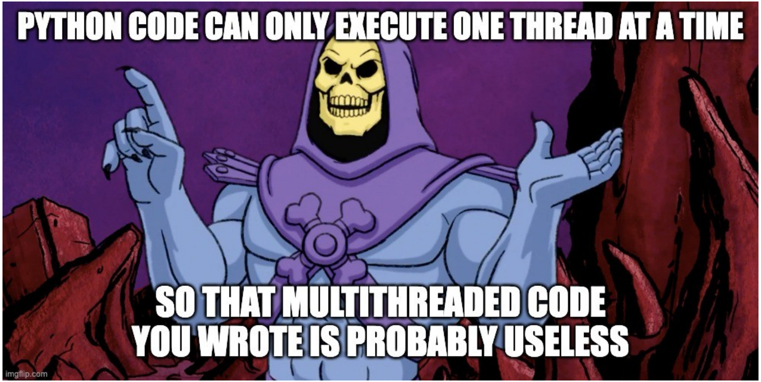
Jim Pivarski

Princeton University – IRIS-HEP

October 23, 2024

What is Python's Global Interpreter Lock (GIL)?

What is Python's Global Interpreter Lock (GIL)?



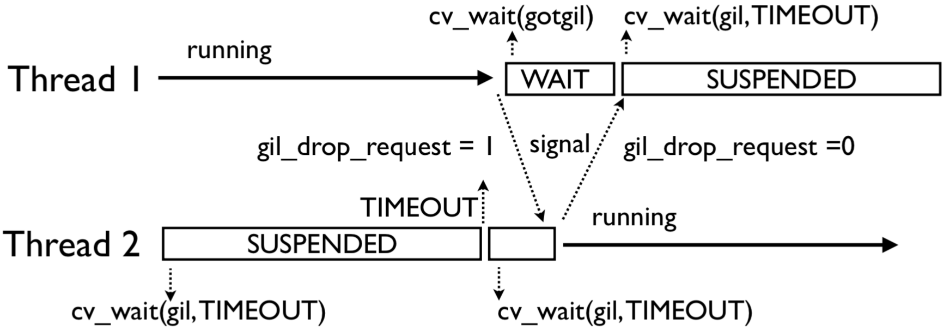
What is Python's Global Interpreter Lock (GIL)?

What is Python's Global Interpreter Lock (GIL)?

In pseudocode:

```
pthread_mutex_lock(&global_interpreter_lock);

PyEval(python_bytecode_instruction);

pthread_mutex_unlock(&global_interpreter_lock);
```

What is Python's Global Interpreter Lock (GIL)?



https://github.com/zpoint/CPython-Internals/blob/master/Interpreter/gil/gil.md

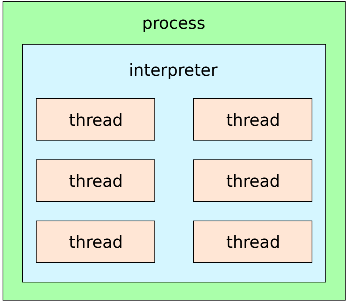
Python 3.13.0 was released 16 days ago.

Python 3.13.0 was released 16 days ago.

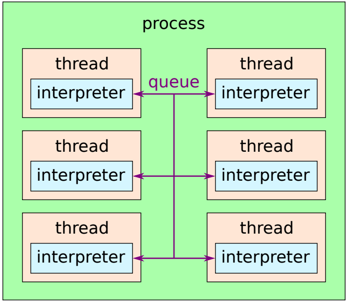
It adds two new ways to avoid the GIL.

Method #1: subinterpreters

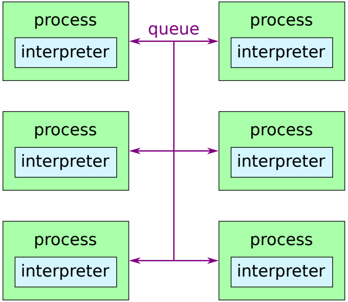
Method #1: subinterpreters



multithreading
all threads can see each other's data
one shared GC & GIL

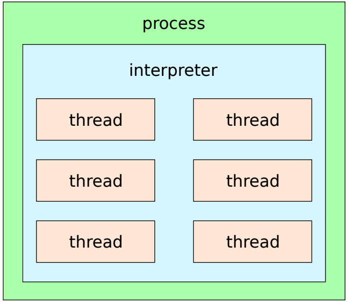


multiple interpreters
each interpreter has its own GC & GIL
looks like one process to the OS
can share array buffers, but not PyObjects

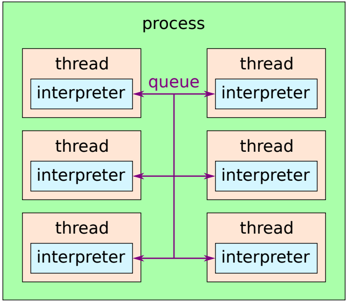


multiprocessing
multiple OS processes
can't share anything without serializing
(or using OS-specific SharedMemory)

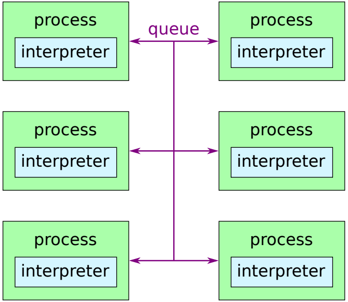
Method #1: subinterpreters



multithreading
all threads can see each other's data
one shared GC & GIL



multiple interpreters
each interpreter has its own GC & GIL
looks like one process to the OS
can share array buffers, but not PyObjects



multiprocessing
multiple OS processes
can't share anything without serializing
(or using OS-specific SharedMemory)

Pre-3.13 trade-off: shared memory + GIL in **multithreading** or shared-nothing + true parallel-processing in **multiprocessing**. Now we have an in-between option.

It is now possible, but not easy, to use subinterpreters in Python

```
from test.support import interpreters
from test.support.interpreters import queues

def in_subinterp():
    # Need to re-import; this is in its own little world...
    from test.support.interpreters import queues

    in_queue = Queue(in_id)        # in_id comes from global scope
    out_queue = Queue(out_id)      # out_id comes from global scope

    x = queue.get()
    out_queue.put(x + number)      # number comes from global scope

in_queue = queues.create()
out_queue = queues.create()

subinterp = interpreters.create()
subinterp.prepare_main({"in_id": in_queue.id, "out_id": out_queue.id, "number": 42})
subinterp.call_in_thread(in_subinterp)

in_queue.put(100)
assert out_queue.get() == 142
```

Very little support from libraries

Many libraries, like NumPy, can't be used in subinterpreters yet.

Very little support from libraries

Many libraries, like NumPy, can't be used in subinterpreters yet.

(NumPy just seg-faults!)

Method #2: free-threading

Method #2: free-threading

```
cd Python-3.13.0/
./configure --disable-gil
make
make install
```

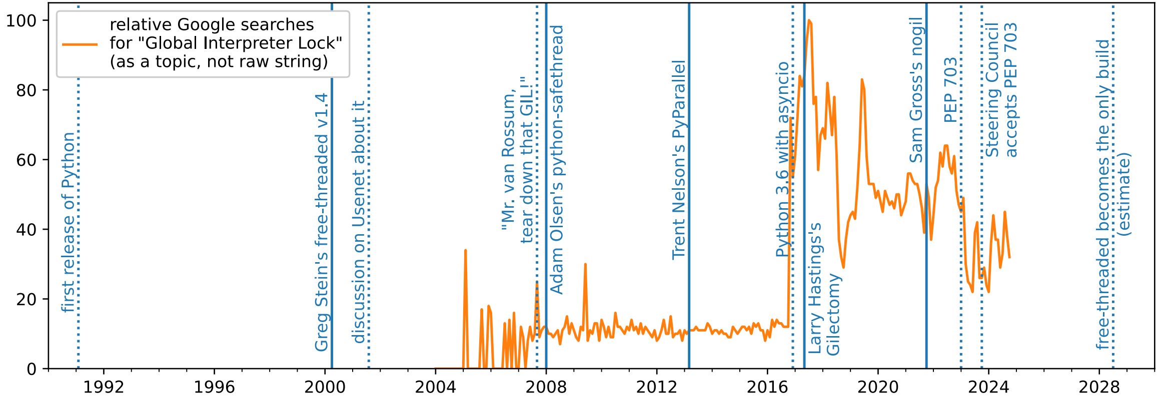
Method #2: free-threading

```
cd Python-3.13.0/
./configure --disable-gil
make
make install
```

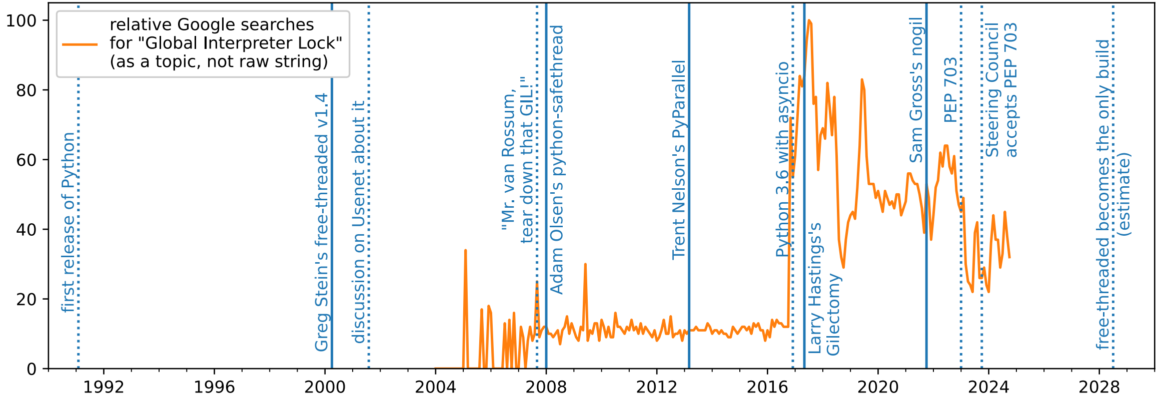
Free-threaded Python is a separate ABI, “`cp313t`”, rather than “`cp313`”.

Compiled extensions have to explicitly opt-in.

Free-threaded Python has been discussed for a long time

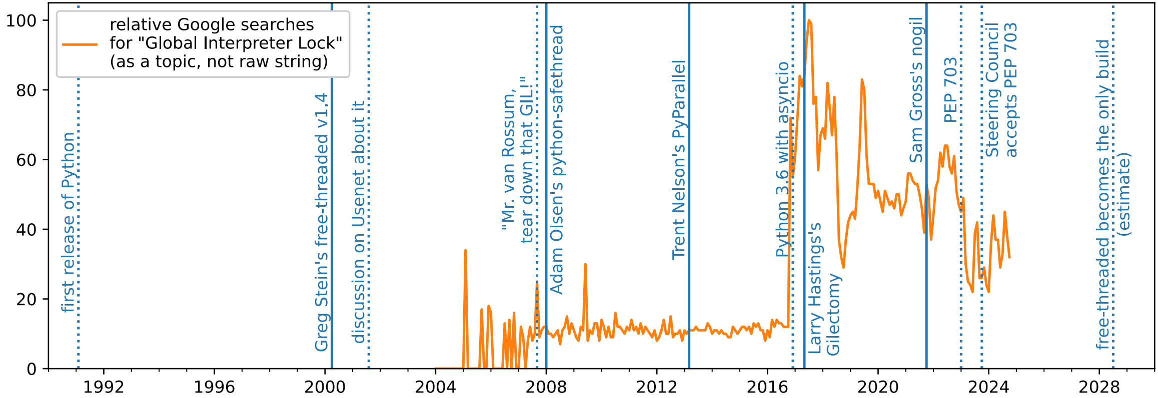


Free-threaded Python has been discussed for a long time



Five forked Pythons, in 2000, 2008, 2013, 2017, 2021, experimentally disabled the GIL.

Free-threaded Python has been discussed for a long time



Five forked Pythons, in 2000, 2008, 2013, 2017, 2021, experimentally disabled the GIL.

Until recently, they all made single (and sometimes multi) threaded performance *worse*.

Why is it working now?

The main issue was CPython's ubiquitous reference counting. Replacing

```
((PyObject*)(obj))->ob_refcnt++;
```

with an atomic operation (or similar) is expensive because it is called so often.

Why is it working now?

The main issue was CPython's ubiquitous reference counting. Replacing

```
((PyObject*)(obj))->ob_refcnt++;
```

with an atomic operation (or similar) is expensive because it is called so often.

J. Choi, T. Shull, J. Torrellas, *Biased reference counting: minimizing atomic operations in garbage collection*, PACT'18 (DOI 10.1145/3243176.3243195).

Why is it working now?

The main issue was CPython's ubiquitous reference counting. Replacing

```
((PyObject*)(obj))->ob_refcnt++;
```

with an atomic operation (or similar) is expensive because it is called so often.

J. Choi, T. Shull, J. Torrellas, *Biased reference counting: minimizing atomic operations in garbage collection*, PACT'18 (DOI 10.1145/3243176.3243195).

Most objects are only referenced by the thread in which they were created.

Why is it working now?

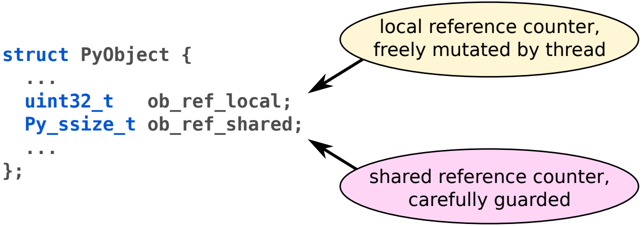
The main issue was CPython's ubiquitous reference counting. Replacing

```
((PyObject*)(obj))->ob_refcnt++;
```

with an atomic operation (or similar) is expensive because it is called so often.

J. Choi, T. Shull, J. Torrellas, *Biased reference counting: minimizing atomic operations in garbage collection*, PACT'18 (DOI 10.1145/3243176.3243195).

Most objects are only referenced by the thread in which they were created.



```
struct PyObject {
  ...
  uint32_t   ob_ref_local;
  Py_ssize_t ob_ref_shared;
  ...
};
```

local reference counter, freely mutated by thread

shared reference counter, carefully guarded

Also. . .

https://peps.python.org/pep-0703

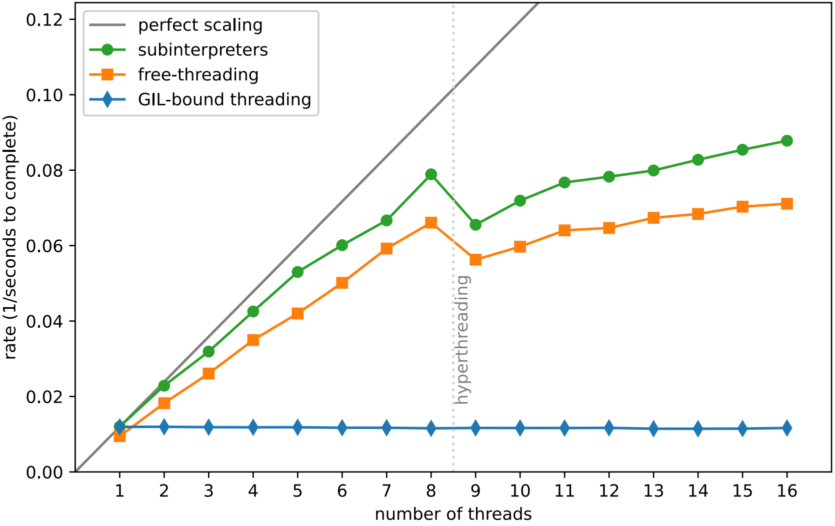
- no reference counting of immortal objects: **`None`**, **`True`**, **`False`**, small integers, interned strings. . .
- deferred reference counting: top-level functions, code objects, modules, methods tend to be accessed by many threads; don't reference count, only garbage collect
- replacing PyMalloc (for small Python objects) with mimalloc
- no linked lists in garbage collecting
- no more generational garbage collecting (reference counting handles short-lived objects)
- locks on all mutable containers (lists, dicts) with optimistic access
- alternatives to borrowed references in C (`PyList_GetItem` → `PyList_FetchItem`)
- "critical sections" in bytecode sequences to avoid deadlocks

Scaling tests

Something computationally expensive in pure Python

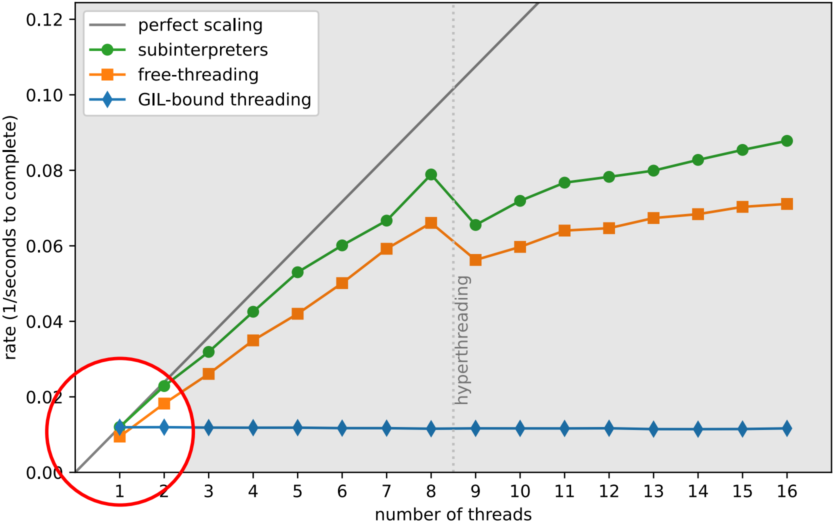
```
# Can't use NumPy in subinterpreters, so use Python's built-in array instead.
offsets = (ctypes.c_int64 * (N + 1)).from_address(ptr_offsets)
pt = (ctypes.c_float * offsets[-1]).from_address(ptr_pt)
eta = (ctypes.c_float * offsets[-1]).from_address(ptr_eta)
phi = (ctypes.c_float * offsets[-1]).from_address(ptr_phi)
mass = (ctypes.c_float * N).from_address(ptr_mass)

# Dimuon mass on all combinations of muons per event...
for event in range(start, stop):
    max_mass = 0
    for i in range(offsets[event], offsets[event + 1]):
        pt1 = pt[i]
        eta1 = eta[i]
        phi1 = phi[i]
        for j in range(i + 1, offsets[event + 1]):
            pt2 = pt[j]
            eta2 = eta[j]
            phi2 = phi[j]
            m = sqrt(2*pt1*pt2*(cosh(eta1 - eta2) - cos(phi1 - phi2)))
            if m > max_mass:
                max_mass = m
    mass[event] = max_mass
```

Scaling test results (8 physical cores)



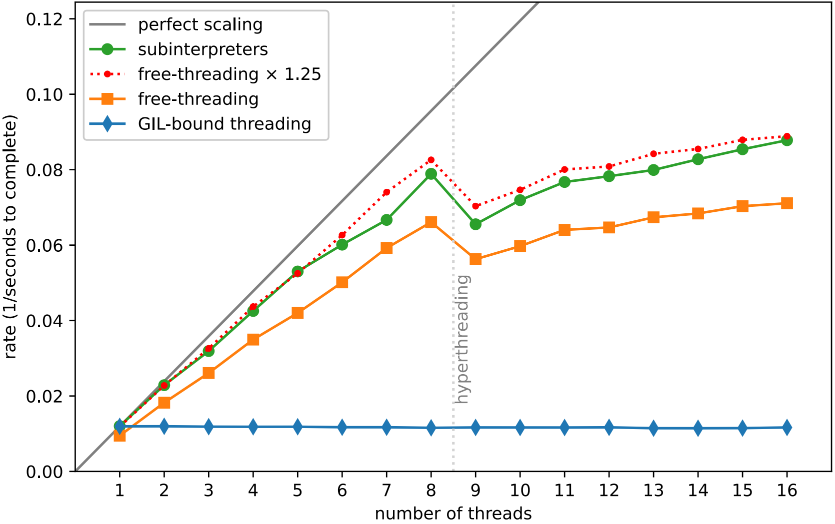
Subinterpreters and free-threading both escape single-thread scaling limit.

Scaling test results (8 physical cores)



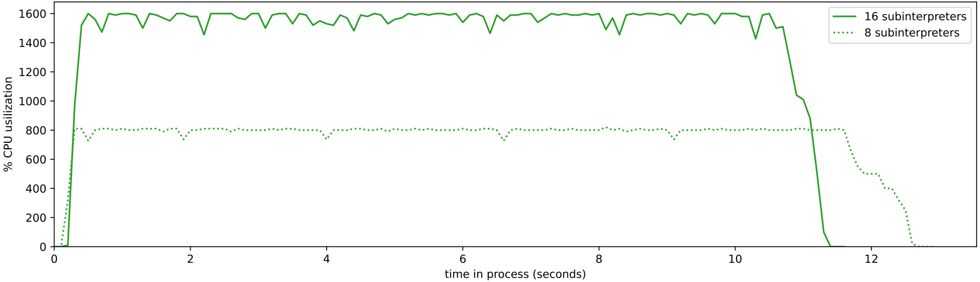
Free-threading doesn’t have all the latest optimizations; single-threaded is slower (for now).

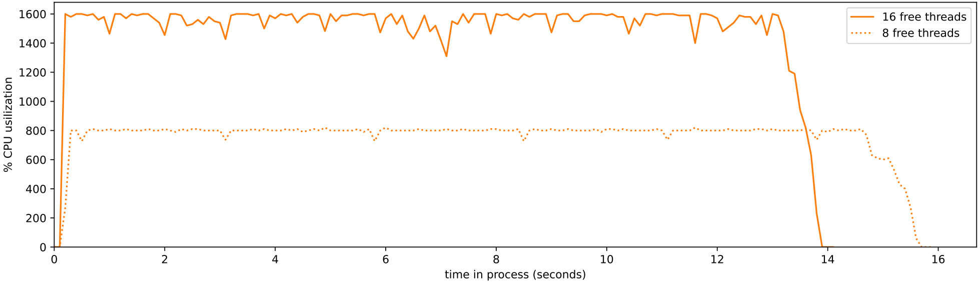
Scaling test results (8 physical cores)



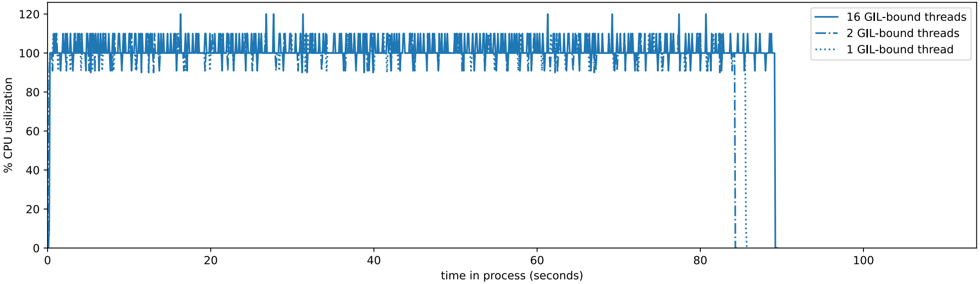
In fact, it's a constant factor.

CPUs are constantly busy, even though scaling isn't perfect



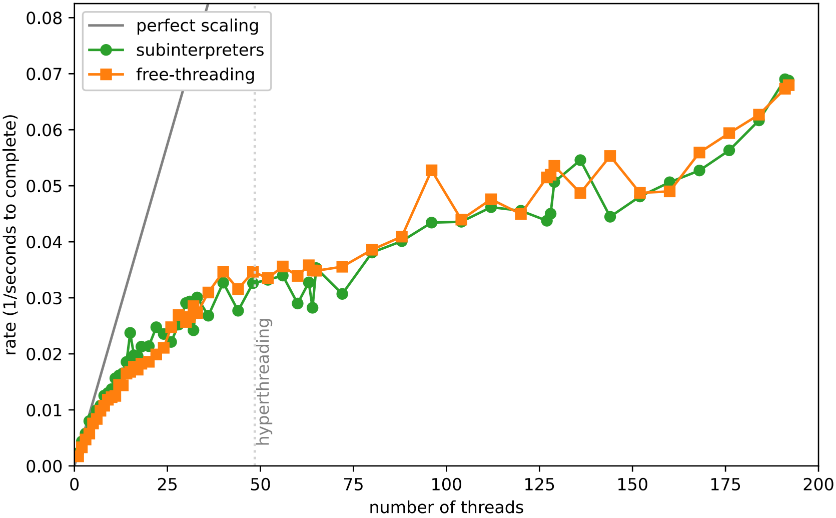


CPUs are constantly busy, even though scaling isn't perfect



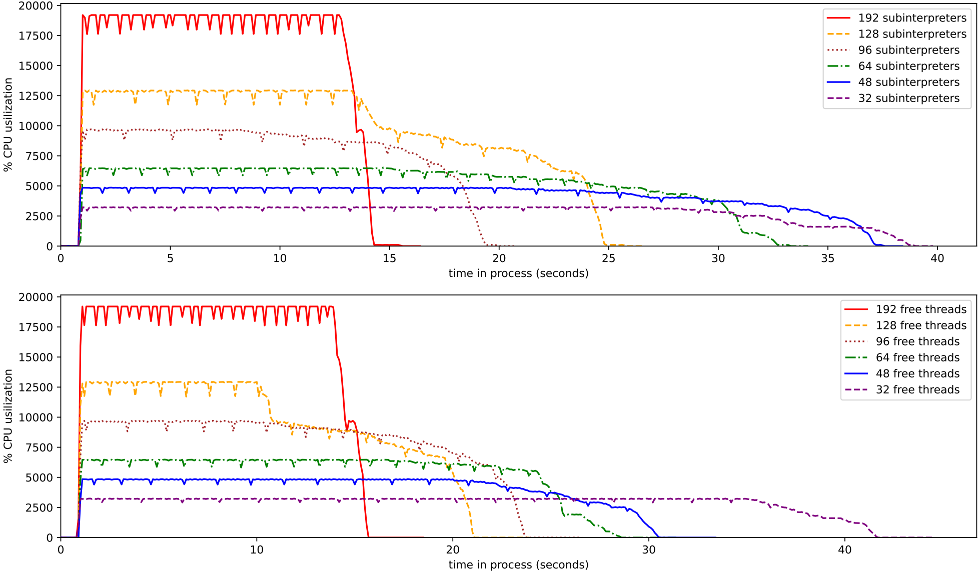
For a pure Python, computationally intensive workload like this, the GIL strictly limits available threads to 1.

Can we go further? (48 physical cores)



The hyperthreading threshold doesn't look significant on this hardware (AWS c7i.metal-48xl).

Not all threads finish equal work in equal times



Scaling tests with Uproot

Uproot has already been (partly) evading the GIL

Most computationally intensive work is offloaded to NumPy and Awkward Array, which release the GIL before numerical computations.

```
Py_BEGIN_ALLOW_THREADS;                   // releases the GIL
big_computation_without_PyObjects();      // other threads run, too
Py_END_ALLOW_THREADS;                     // re-acquires the GIL
return result_with_PyObjects;
```

Uproot has already been (partly) evading the GIL

Most computationally intensive work is offloaded to NumPy and Awkward Array, which release the GIL before numerical computations.

```
Py_BEGIN_ALLOW_THREADS;                  // releases the GIL
big_computation_without_PyObjects();     // other threads run, too
Py_END_ALLOW_THREADS;                    // re-acquires the GIL
return result_with_PyObjects;
```

But we only enter GIL-released C code on a per-TBasket basis.

The code between these excursions are synchronization points (Amdahl's law).

Two ways to parallelize Uproot

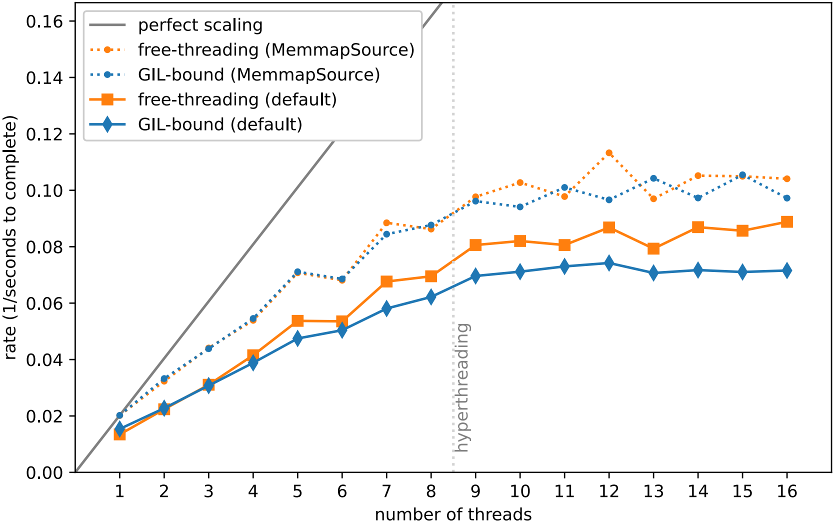
"External": some code that controls threading (e.g. Dask) calls Uproot

```
def in_thread(uproot_tree, start, stop):
    return uproot_tree.arrays(entry_start=start, entry_stop=stop)

executor = ThreadPoolExecutor(max_workers=N)
batches = executor.map(in_thread, list_of_args_tuples)
```

"Internal": Uproot reads TBaskets in parallel but returns one array

```
executor = ThreadPoolExecutor(max_workers=N)

array = uproot_tree.arrays(
    decompression_executor=executor, interpretation_executor=executor
)
```

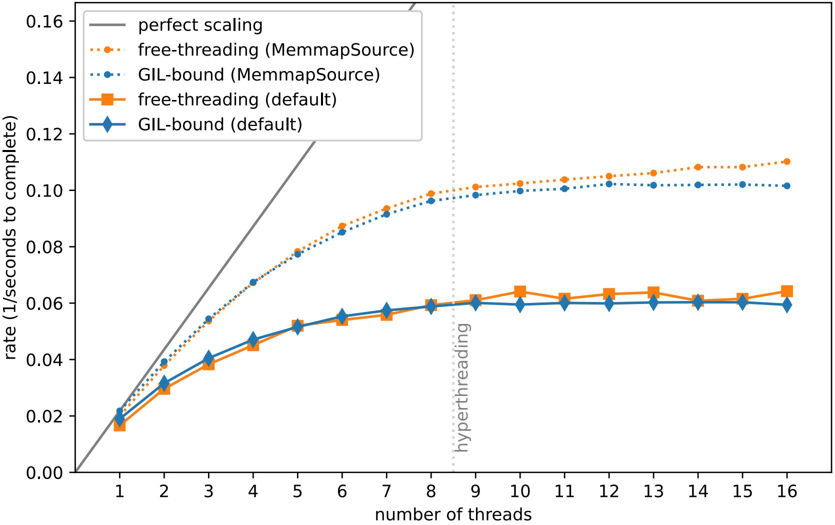
Parallelizing Uproot "externally"



GIL-bound is not bad, but there's a small improvement.

The bigger difference is between the default file `handler` and `MemmapSource`.

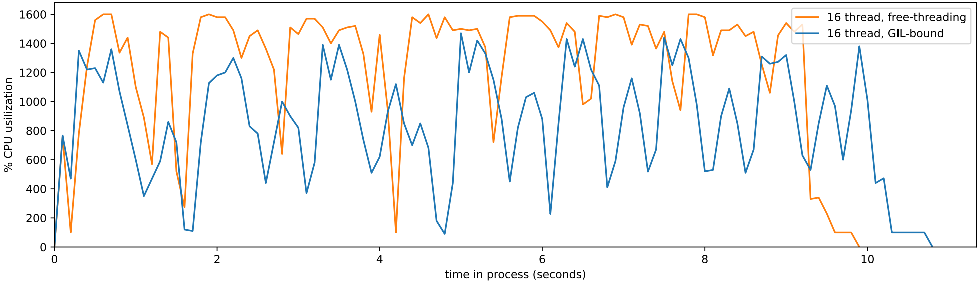
Parallelizing Uproot “internally”



GIL-bound is not bad, but there's a small improvement.

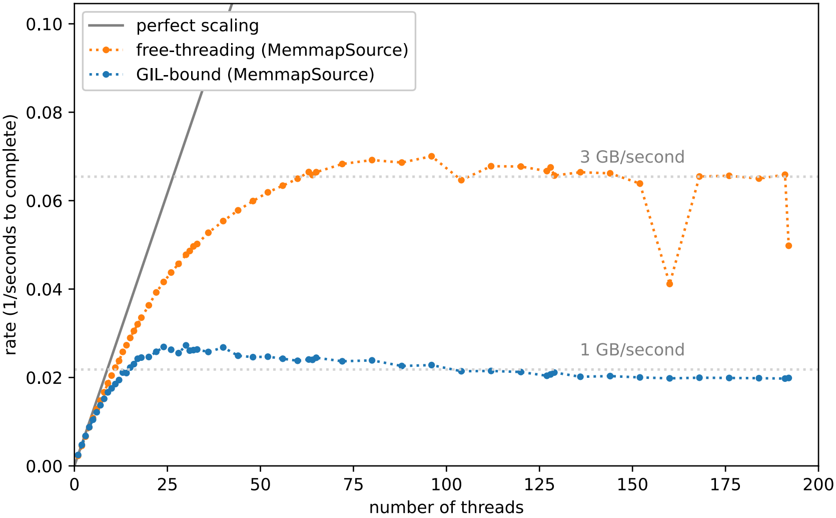
Especially in the internal case (more fine-grained; less waste from multiple threads reading the same TBaskets).

CPUs are not always busy, but free-threaded is busier. . .



Note: file-reading tasks performed with warm cache, so RAM → CPU is the only I/O.

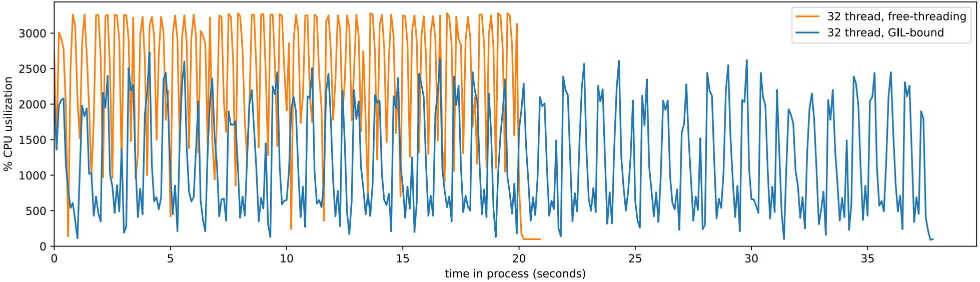
Can we go further? (48 physical cores, “internal” parallelization)

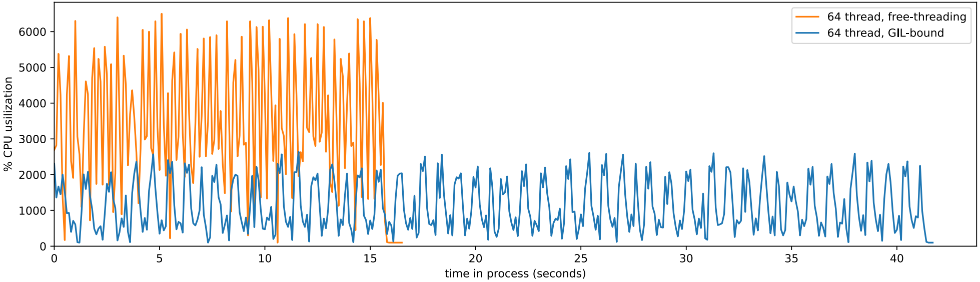


Free-threading starts to be relevant above 8 threads and keeps getting better until 3 GB/second.

You need well over 8 cores to see this.

CPUs are still not always busy, but free-threaded is busier. . .





Conclusions

- Python 3.13 provides two new ways to avoid the GIL.

Conclusions

- Python 3.13 provides two new ways to avoid the GIL.
- **Subprocessors** require more effort from Python users and are not well supported by libraries (NumPy).

Conclusions

- Python 3.13 provides two new ways to avoid the GIL.
- **Subprocessors** require more effort from Python users and are not well supported by libraries (NumPy).
- **Free-threading** required a massive overhaul of Python's internals, but "just works" from a Python user's perspective.

Conclusions

- Python 3.13 provides two new ways to avoid the GIL.
- **Subprocessors** require more effort from Python users and are not well supported by libraries (NumPy).
- **Free-threading** required a massive overhaul of Python's internals, but "just works" from a Python user's perspective.

  (Python community is much more interested in free-threading.)

Conclusions

- Python 3.13 provides two new ways to avoid the GIL.
- **Subprocessors** require more effort from Python users and are not well supported by libraries (NumPy).
- **Free-threading** required a massive overhaul of Python's internals, but "just works" from a Python user's perspective.

  (Python community is much more interested in free-threading.)
- They scale identically, apart from a constant factor (bytecode optimizations, to be implemented later in free-threading mode).

Conclusions

- Python 3.13 provides two new ways to avoid the GIL.
- **Subprocessors** require more effort from Python users and are not well supported by libraries (NumPy).
- **Free-threading** required a massive overhaul of Python's internals, but "just works" from a Python user's perspective.

  (Python community is much more interested in free-threading.)
- They scale identically, apart from a constant factor (bytecode optimizations, to be implemented later in free-threading mode).
- Uproot has already been releasing the GIL, but benefits from free-threading if you have a lot more than 8 cores.