Circumventing the Limitations of Python

Samuel Yang-Zhao

National Computational Infrastructure, Australia

Acknowledgement of Country

The National Computational Infrastructure acknowledges, celebrates and pays our respects to the Ngunnawal and Ngambri people of the Canberra region and to all First Nations Australians on whose traditional lands we meet and work, and whose cultures are among the oldest continuing cultures in human history.

Outline

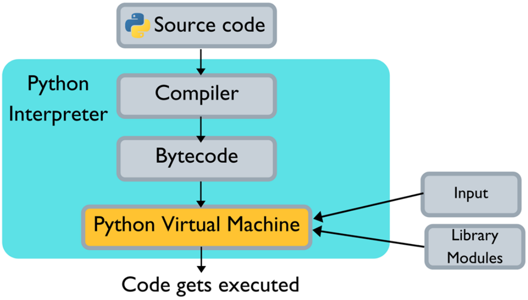
Python is one of the most popular programming languages today. Whilst code written in Python can be slow, in this course we will investigate ways in which we can circumvent Python's limitations.

We will cover:

- How Python works under the hood
- What is the GIL?
- Circumventing the GIL with the Python Multiprocessing Module
- Introduction to Cython
- Turning off the GIL and parallel computing in Cython
- Wrapping C/C++ using Cython

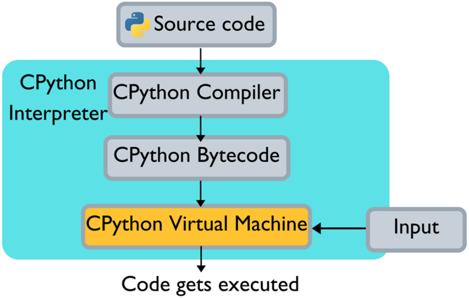
Python under the hood

Python is an interpreted language. The Python interpreter first compiles the source code into platform-independent Bytecode before it is executed on the Python Virtual Machine.



Python under the hood

The standard reference implementation of Python is CPython. It implements the Python interpreter using both C and Python. CPython's memory management is not *thread-safe*, i.e. no guarantees against race conditions.

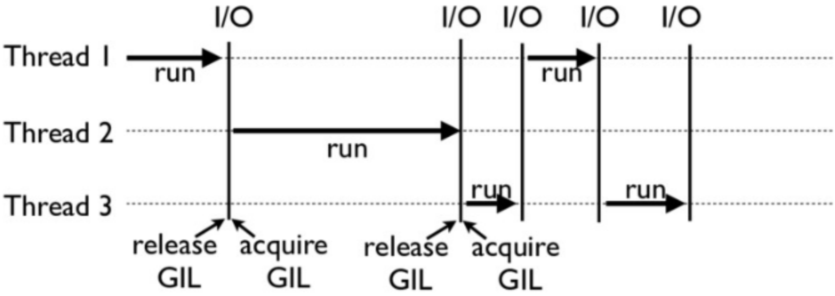


What is the GIL?

The Global Interpreter Lock, or GIL, is CPython's solution to prevent multiple native threads from causing unwanted interactions. Essentially, the GIL is one big mutex allowing only one thread to execute the Bytecode at any given time.

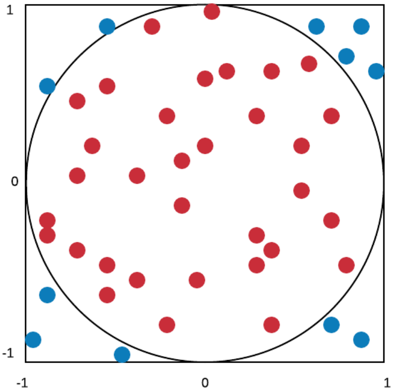
What is the GIL?

The Global Interpreter Lock, or GIL, is CPython's solution to prevent multiple native threads from causing unwanted interactions. Essentially, the GIL is one big mutex allowing only one thread to execute the Bytecode at any given time.



What about Python's threading Module?

Consider computing π by Monte-Carlo simulation. We sample points uniformly from $[-1,1]^2$ and the probability of a point x landing in the unit circle is $\mathbb{P}(x \in \text{circle}) = \frac{\pi}{4}$. Thus $\pi \approx 4 \times \frac{\text{samples in circle}}{\text{total samples}}$.



What about Python's threading Module?

Clearly, using threads did not help and was actually detrimental! The extra run-time can be attributed to the extra overhead incurred in switching between and managing threads whilst parallel processing is prevented by the GIL.

Method	**Time**
Serial computation	$\sim 11s$
Using 4 threads	$\sim 26s$

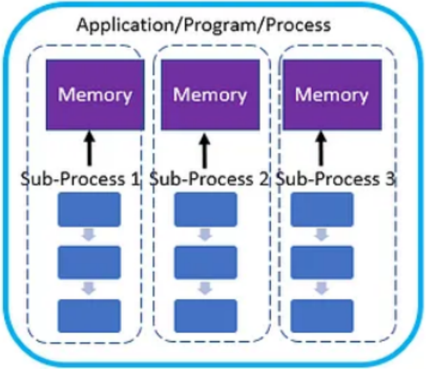
Why have the GIL at all?

So far we have seen that the GIL can prevent multithreading from boosting performance. So why did Python choose to use a GIL?

- The alternative is fine-grained locking, i.e. locking only on variables/data that should not be concurrently accessed. This can cause slower single-threaded performance.
- For I/O bound programs, the multi-threaded performance is faster.
- Multi-threaded performance is also faster when compute-intensive workload is handed off to C libraries.
- Makes writing C libraries simple. Worried about thread-safety? Keep GIL on.

Python's `multiprocessing` Module

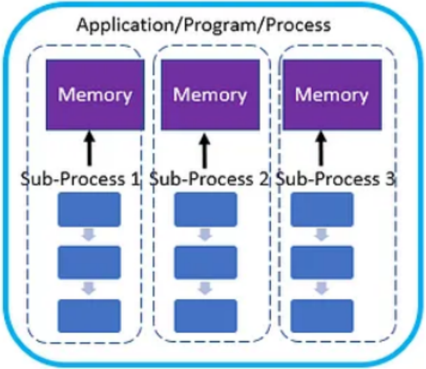
Python's multiprocessing module allows you to spawn multiple native processes within a program. Each process is allocated its own block of memory.



Python's `multiprocessing` Module

Python's multiprocessing module allows you to spawn multiple native sub-processes within a program. Each sub-process is allocated its own memory.

- Each process can run on different CPU cores.
- Can sidestep the GIL.
- Slow inter-process communication.
- More resource intensive than threads.



Using `multiprocessing.Process`

The `multiprocessing.Process` class can be used to initialize subprocesses.

```python
import multiprocessing as mp
def f(name):
    print(f"Hello {name}")

p = mp.Process(target=f, args=('bob',)) # args must be a tuple
p.start() # start process
p.join()  # wait for process to complete
```

Using `multiprocessing.Process`

There are no direct ways to return a value from a `mp.Process`. Instead we can instantiate a `mp.Queue` object to store outputs.

```
import multiprocessing as mp
import random
def rand_integers(n, q):
    q.put([random.randint(0, 10) for i in range(n)])

N = 100
q = mp.Queue()
p = mp.Process(target=rand_integers, args=(N,q))
p.start()
p.join()
output = q.get()
```

Using `multiprocessing.Process`

Multiple processes can be instantiated and then executed in a loop.

```
nprocesses = 4
processes = []
for i in range(nprocesses):
    processes += [mp.Process(target=rand_integers, args=(N,q))]
for p in processes:
    p.start()
for p in processes:
    p.join()
output = [q.get() for p in processes]
```

Using `multiprocessing.Pool`

Another more convenient approach for simple parallel processing is to use the `multiprocessing.Pool` class. `Pool` provides four basic methods for parallel processing:

- `Pool.apply`
- `Pool.map`
- `Pool.apply_async`
- `Pool.map_async`

Which one you choose depends upon whether you need multiple arguments, concurrency, blocking and ordering.

Using `multiprocessing.Pool`

`Pool.apply` takes a function and applies it to the given arguments.

```
def cube(x):
    return x**3

pool = mp.Pool(processes=4)
results = [pool.apply(cube, args=(x,)) for x in range(1,7)]
```

`Pool.apply` blocks the main program's execution until the result completes and hence not suitable for parallel execution. However it will return results in order and can take multiple arguments.

Using `multiprocessing.Pool`

`Pool.map` takes a function and an iterable and applies the function to each element of the iterable.

```
...
pool = mp.Pool(processes=4)
results = pool.map(cube, range(1,7))
```

`Pool.map` also blocks the main program's execution until the result completes. It can still perform parallel computations as the `Pool.map` does not block internally. The results are returned in order and it can only take one iterable argument.

Using `multiprocessing.Pool`

`Pool.apply_async` and `Pool.map_async` are called identically to `Pool.apply` and `Pool.map` but require a call to `.get` to retrieve the output.

```
...
pool = mp.Pool(processes=4)
output = [pool.apply_async(cube, args=(x,)) for x in range(1,7)]
results = [p.get() for p in output]
...
output = pool.map_async(cube, range(1,7))
results = output.get()
```

Both methods execute asynchronously and do not block the main program's execution. The output ordering is no longer guaranteed.

Using `multiprocessing.Pool`

In summary, we have

Method	Multiple args	Blocking	Concurrency	Ordering
`Pool.apply`	Y	Y	N	Y
`Pool.map`	N	Y	Y	Y
`Pool.apply_async`	Y	N	Y	N
`Pool.map_async`	N	N	Y	N

Exercise

Work through `multiprocessing.ipynb`.

Introduction to Cython

Cython is a programming language that makes writing C/C++ extensions for the Python language easy. We will consider the workflow whereby our main program is written in Python and we wish to hand off costly computations to Cython to be more efficient.

We will cover:

- Cython language basics
- Releasing the GIL in Cython
- Wrapping C/C++ using Cython

Note: We will assume some basic familiarity with C.

Hello world

helloworld.pyx

```
print("hello world")
```

setup.py

```
from setuptools import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

ext=[Extension(name='helloworld',
     source=['helloworld.pyx'],)]

setup(ext_modules = ext,
      cmdclass = {'build_ext': build_ext},)
```

- Cython code is written in .pyx files.
- The `Extension` class is used to describe our Cython extension.
- The `name` argument is name of the module when imported into Python.
- The source argument is a list of source files.
- `setup()` is used to build the Cython extension into an importable module.

Compiling Hello world

Our directory initially contains just our .pyx file and the setup.py

```
(ML) u6642247@siiml-01:~/Python-HPC/part2/demo3$ ls -l
total 8
-rw-r--r-- 1 u6642247 users  23 Mar 22 16:31 helloworld.pyx
-rw-r--r-- 1 u6642247 users 326 Mar 22 16:31 setup.py
```

We first compile our Cython extension with the following command.

```
$ python helloworld_setup.py build_ext --inplace
```

Compiling produces a build directory, a shared object file (`.so` file) and a .c source code file. The `-inplace` flag moves the `.so` file from the build directory into the current directory.

```
(ML) u6642247@siiml-01:~/Python-HPC/part2/demo3$ ls -l
total 144
drwxr-xr-x 3 u6642247 users   4096 Mar 22 16:32 build
-rw-r--r-- 1 u6642247 users 109980 Mar 22 16:32 helloworld.c
-rwxr-xr-x 1 u6642247 users  23968 Mar 22 16:32 helloworld.cpython-310-x86_64-linux-gnu.so
-rw-r--r-- 1 u6642247 users     23 Mar 22 16:31 helloworld.pyx
-rw-r--r-- 1 u6642247 users    326 Mar 22 16:31 helloworld_setup.py
```

The .so file is the module that we can import into Python. We can now import our Cython extension as a module.

```python
import helloworld # prints "hello world"
```

Declaring C variables and functions

C/C++ variables and functions can be declared in Cython using `cdef`. As in C, the type of the variable must also be declared.

```
# C variables
cdef int a_global_variable = 1
cdef double x = 5.0

# Python f'n with C variables
def py_func(int i, char *r):
    ...

# C f'n
cdef int c_func(int x, int y):
    ...

# C f'n with Python args
cdef int mix_func(x, y):
    ...
```

- Cython can compile both Python code and also C/C++ code written using `cdef`.
- Only Python functions in a Cython module can be called by Python scripts.
- Any variable that is undeclared is treated as a Python object.
- Ideally we want to avoid Python objects as much as possible.

Visualising Python references

Cython comes with a handy tool to visualise which lines of code require references to Python. Running `\$ cython script.pyx -a` on the terminal gives a html file when opened which highlights where the code accesses Python:

```
Generated by Cython 0.29.21

Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C code that Cython generated for it.

Raw output: demo3.c

+01: cdef int mix_func(x):
+02:     cdef int y = 5+x
+03:     return y
 04:
+05: cdef int c_func(int x):
+06:     cdef int y = 5+x
+07:     return y
 08:
+09: def func(y):
+10:     return c_func(y)
```

Built-in C libraries and typecasting

Cython provides access to the standard C libraries (`stdlib.h`) without adding any extra dependencies.

```
from libc.stdlib cimport rand, RAND_MAX

def random_uniform():
    return rand() # return integer between 0 and RAND_MAX
```

Linking to shared object libraries

Some libraries however, will require linking to the shared library. For example, we can import the `math.h` in Cython.

`cy_math.pyx`

```
from libc.math cimport sin
cdef double sin_squared(double x):
    return sin(x * x)
```

On some systems, we will need to let the compiler know that it needs to link against the shared library `m`. Using `setuptools`, it is sufficient to add it to the `libraries` argument in `Extension`:

`setup.py`

```
ext_modules = [
    Extension(name="cy_math", sources=['cy_math.pyx'], libraries=["m"])
]
...
```

Exercise

Work through Part 1 of `cython.ipynb`

Cython provides native parallelism support through the `cython.parallel` module. This module provides parallelization using the OpenMP constructs `omp parallel for` and `omp parallel` under the hood. These are ideal for parallelizing relatively small, self-contained blocks of code. To use the parallelism provided, the GIL must also be released.

Parallel for loops with `prange`

To parallelise loops, we can use `prange`.

```
from cython.parallel cimport prange

cdef int i
cdef int n = 200
cdef int sum_ = 0

for i in prange(0, n, nogil=True):
    sum_ += i
```

- `prange` operates similarly to Python's `range` function.
- The first two arguments indicate the starting index and the stopping index (does not include stopping index).
- `prange` can only be run with **`nogil`** must be set to True.
- When **`nogil`** is true, the whole loop is treated as being run inside a **`nogil`** section.

Parallel for loops with `prange`

If we wish to call a function within `prange`, the function must be declared with **`nogil`**.

```
from cython.parallel cimport prange
from libc.math cimport sin

cdef double sin_squared(int x) nogil:
    return sin(x*x)

cdef int i
cdef double X[200]
for i in prange(200, nogil=True):
    X[i] = sin_squared(i)
```

Compiling `cython.parallel`

Since `cython.parallel` uses OpenMP, we need to tell the compiler to enable OpenMP. We can do this by providing compiler arguments and link arguments to the compiler in the Extension object in `setup.py`.

```
ext_modules=[
    Extension("dice6_cy3",
              ["dice6_cy3.pyx"],
              libraries=["m"],
              extra_compile_args = ["-fopenmp"],
              extra_link_args=['-fopenmp']
              )
]
```

Note: Other compile and link arguments can be provided in extra_compile_args and extra_link_args.
e.g. `"--fast-math"` can be provided in the list as an extra compile argument.

Wrapping C/C++

Suppose we have the following code written in C++ for sampling and counting points that land in the unit square.

pi.h

```
int sample(void);
double sin_squared(int x);
```

pi.cpp

```
#include <stdlib.h>
#include "pi.h"
int sample(void){
    double x = ((double) rand()) / (RAND_MAX);
    double y = ((double) rand()) / (RAND_MAX);
    if ((x*x + y*y) <= 1) return 1;
    return 0;
}
```

Cython can help provide an interface between Python and our C++ code.

Wrapping C/C++

We need to include the C++ code and expose which functions from the header file we wish to use in the Cython code. These declarations are placed in a Cython header file (a .pxd file). This then exposes the C++ functions when imported into a .pyx file.

pi_cython.pxd

```
cdef extern from "pi.cpp":
    pass
cdef extern from "pi.h":
    int sample()
```

pi_cython.pyx

```
from pi_cython cimport sample
def cy_serial_sampler(n):
    cdef int i
    cdef int count = 0
    for i in range(n):
        count += sample()
    return count
```

Wrapping C/C++

To compile, we now need to specify that the language our code is in is C++. We can also specify the C++ standard we wish to use as a compile argument.

```
ext_modules = [
    Extension(name = "pi_cython",
              language='c++',
              extra_compile_args=['-std=c++17'],
              sources=["pi_cython.pyx"]
    )
]
```

NOTE: Name your .pyx file differently from your .cpp file as the setup.py compilation process will produce a .cpp file of the same name as your .pyx file, overwriting your original .cpp file.

Exercise

Complete Part 2 of `cython.ipynb`