Breaking free from the GIL

Code: `github.com/sueszli/nogil`

In October 2024, Python surpassed JavaScript for the first time as the most popular language on GitHub's Octoverse[1]. This high-level, interpreted, garbage-collected and dynamically-typed language emphasizes code readability and simplicity both in its implementation and syntax. This emphasis on simplicity also reflects itself in the language's community, often referred to as the "Pythonic" way of doing things.

However, the language's simplicity comes at a cost: While memory and network-bound tasks can be efficiently managed through colored functions[2] and `asyncio`, Python is notoriously slow for compute-bound tasks due to the Global Interpreter Lock (GIL)[3]. The GIL is a mutex in Python's most popular implementation, CPython, that protects access to Python objects, preventing multiple threads from executing Python bytecodes simultaneously and therefore limits the language's performance on multi-core systems.

Or as Rob Pike put it in 2012[4]:

> *"The computing landscape today is almost unrelated to the environment in which the languages being used, mostly C++, Java, and Python, had been created. The problems introduced by multicore processors, networked systems, massive computation clusters, and the web programming model were being worked around rather than addressed head-on. Moreover, the scale has changed: today's server programs comprise tens of millions of lines of code, are worked on by hundreds or even thousands of programmers, and are updated literally every day. To make matters worse, build times, even on large compilation clusters, have stretched to many minutes, even hours."*

These limitations have led to the development of various workarounds, such as the development of competing superset languages such as the `taichi`[5] and `mojo`[6], optimizing Python interpreters like `PyPy`[7] and `Numba`[8], proposing to introduce multiple lightweight sub-interpreters[9] [10], or even making the GIL entirely optional[11] [12] [13], as proposed in PEP 703[14].

The latter approach, making the GIL optional, was recently accepted in Python 3.13 and is currently in the experimental stage. There is no guarantee of this feature being included in the final release, but it is a step in the right direction.

In the past, parallelizing Python code was primarily achieved through the `multiprocessing` module, which, while effective, comes with significant drawbacks: it lacks shared memory support and consumes more resources compared to threading.

The introduction of a GIL-free Python has raised hopes for more efficient parallelization. However, it's critical to recognize that Python's inherently dynamic nature and the overhead of its interpreter limit its performance far beyond the GIL. Even with these improvements, Python remains significantly slower - often by a factor of 1000 or more - compared to statically-typed systems languages[15].

This begs the question: why has Python become so dominant in scientific computing? The answer lies in its role as a high-level orchestration tool. In this domain, computational bottlenecks are offloaded to libraries written in optimized languages like C or Fortran, while Python acts as a glue language. This design is so entrenched that systems engineers frequently use macros to embed C code directly into Python[16].

Recent developments in Python's concurrency capabilities, such as the removal of the GIL, are therefore not a revolution in raw performance but an enhancement in convenience. They make it easier for developers to write efficient code, especially for tasks that can benefit from parallel execution, without fundamentally altering Python's comparative speed limitations.

---

[1] https://github.blog/news-insights/octoverse/octoverse-2024/#the-most-popular-programming-languages
[2] https://langdev.stackexchange.com/questions/3430/colored-vs-uncolored-functions
[3] Wang, Z., Bu, D., Sun, A., Gou, S., Wang, Y., & Chen, L. (2022). An empirical study on bugs in python interpreters. IEEE Transactions on Reliability, 71(2), 716-734.
[4] https://go.dev/talks/2012/splash.article
[5] https://www.taichi-lang.org/
[6] https://docs.modular.com/mojo/stdlib/python/python.html
[7] https://www.pypy.org/
[8] https://numba.pydata.org/
[9] https://peps.python.org/pep-0554/
[10] https://peps.python.org/pep-0683/
[11] https://peps.python.org/pep-0703/
[12] https://discuss.python.org/t/a-steering-council-notice-about-pep-703-making-the-global-interpreter-lock-optional-in-cpython/30474
[13] https://engineering.fb.com/2023/10/05/developer-tools/python-312-meta-new-features/
[14] https://peps.python.org/pep-0703/
[15] https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html
[16] https://docs.python.org/3/c-api/init.html#releasing-the-gil-from-extension-code

In this report, we will explore the optimization of a compute-bound task in Python across varying levels of abstraction. We will analyze the trade-offs between performance and usability, shedding light on how Python's evolving capabilities can be leveraged effectively.

1. Algorithm: Naive Brute-Force Collision Attack

The embarrassingly parallel algorithm we have selected to demonstrate our optimizations, particularly the GIL-free multithreaded implementation, to put in contrast with the alternatives is a brute-force password cracker, most popular from the open-source library `hashcat`. The algorithm is simple: given a target hash, from a given character set and a maximum password length, the cracker generates all possible passwords and hashes them using the same algorithm. If the generated hash matches the target hash, the password is found and the program terminates.

To formalize, we define: character set C, maximum password length n, target hash value h_{target}, and hash function $H(x)$.

The search space S can be expressed as $S = \bigcup_{i=1}^{n} C^i$ where C^i represents all possible strings of length i from character set C. The problem can then be formulated as finding $p \in S$ such that: $H(p) = h_{target}$ and the total search space size is $|S| = \sum_{i=1}^{n} |C|^i$.

For parallel processing, we can partition S into k subsets: $S = \bigcup_{j=1}^{k} S_j$ where each S_j can be processed independently by different threads.

To simplify the problem, we have chosen a small character set $C = \{a, \dots z\}$ and a small maximum password length $n = 8$. The target hash value is set to `aaa` for simplicity. More sophisticated attacks would also consider the possibility of salted hashes, which we have omitted for brevity as well as clever algorithmic optimizations like bloom filters, rainbow tables, etc. which are beyond the scope of this report.

In initial experiments, we self-implemented 3 distinct hash functions $H(x)$ in Python, namely md5, sha1, and sha256, in plain Python to compare both the simplicity and security of the hash functions against our "collision attack". It's worth noting that none of these hash functions are considered secure for password hashing[17].

- `md5`: ~100 LoC, max 9847.03 hashes per second
- `sha1`: ~40 LoC, max 18578.26 hashes per second (approx. half as fast as the hashlib implementation)
- `sha256`: ~50 LoC, max 7870.21 hashes per second

To validate the correctness of our implementations, we have cross-validated the results with the `hashlib` library in Python. Due to the slow performance of any implementation other than our thoroughly optimized `sha1` implementation, we have decided to use it for the remainder of our experiments - unless otherwise noted.

2. Methodology: Different Levels of Abstraction

To benchmark our experiments with the experimental GIL-free CPython version 13t (threaded), we implemented a Docker container that builds Python from source and sets the correct compile flags to disable the GIL. The container additionally serves to increase isolation and the reproducibility of our experiments.

1) Plain Python The most straightforward implementation of the brute-force attack is a simple loop that iterates over all possible passwords and hashes them. We have implemented this in three different ways: (1) completely free of any libraries, (2) using the `itertools` library to improve succinctness and (3) using the `hashlib` library for hashing (in just 4 LoC).

2) Multiprocessing The `multiprocessing` module in Python allows for parallel processing by creating separate processes for each task. We have implemented the brute-force attack using the `map`, `imap`, `map_async`, and `imap_unordered` functions to compare the performance of different parallelization strategies. We omitted the `starmap` function as it is just syntactic sugar for `map` and the Executor API as the overhead of creating a new process for each task is too high.

When using the `multiprocessing` library in Python, we can call multiple system processes that each come with their own separate Python interpreter, GIL, and memory space. This is very simple, straightforward, and the intended way to write parallel code in the latest Python version. But it comes with all the pros and cons of using processes for parallel programming:

- Simple, has higher isolation, security and robustness.
- Context switching: actually doesn't matter, since the threading library threads are kernel-level as well.
- Resource overhead: memory allocation, creation, and management are slower for processes. Additionally, having a unique copy of the interpreter for each process is really wasteful.

[17] https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

- Serialization overhead: there is no shared memory, so data has to be serialized and deserialized for inter-process communication. Also, some objects are unserializable: the pickle module is used to serialize objects. But some objects are not pickleable (i.e. lambdas, file handles, etc.).

3) Multithreading The `threading` module in Python allows for parallel processing by creating separate threads for each task. With the GIL-free Python, this level of abstraction is expected to experience the most significant performance improvements. We have implemented the brute-force attack using the `ThreadPoolExecutor` and `Thread` classes to compare the performance of different parallelization strategies. While the `ThreadPoolExecutor` is more convenient, the `Thread` class allows for more fine-grained control over the threads and resembles the join API of C more closely.

When using the `threading` library in Python with the GIL released using the `GIL=0` flag, we achieve the lowest overhead and the highest performance of any pure Python implementation for parallel processing.

4) ctypes CTypes is a foreign function interface (FFI) for Python that allows calling functions from shared libraries.

We initially implemented our own optimized Sha1 implementation in C, but noticed that it performs almost identically to the `openssl/sha.h` implementation. We have therefore decided to use the more robust implementation instead to reduce complexity in our experiments. We conducted two experiments: one in plain C and one using OpenMP to parallelize the hashing function.

This method is the most straightforward way to call C code from Python and requires close to no knowledge of the Python C API[18]. The GIL is released automatically on each foreign function call. However, it comes with massive serialization overhead as automatic type conversions done by the FFI-library are very expensive. This can be partially circumvented by passing pointers or using CFFI but it will still be significantly slower than extending CPython. Ideally one should share as little data, pass as little data and call the foreign function as little as possible.

We did so by only passing the target hash h_{target} to the C function and returning the password if it matches.

Ctypes aren't meant to be used for high performance libraries that you use frequently but codebase-glue. you can still use them for that purpose and gain a significant amount of performance, but you have to move as much of the computation as possible into the C implementation.

5) CPython Finally, we have implemented the brute-force attack by extending the CPython interpreter directly. Given our lessons from previous experiments, we have decided to use the `openssl/sha.h` implementation for hashing and leave out the OpenMP parallelization.

Extending CPython has neglible to no overhead and allows to share large chunks of memory by calling `mmap()` directly. However, it comes with a very complex API and requires you to manually manage the GIL with `Py_BEGIN_ALLOW_THREADS` and `Py_END_ALLOW_THREADS` macros and marshal all data passed between the C and Python code. It also isn't portable and requires a lot of boilerplate code to build and distribute.

3. Results

We achieved a ~6x speedup of the plain python implementation, even marginally outperforming the `hashlib.sha1` library.

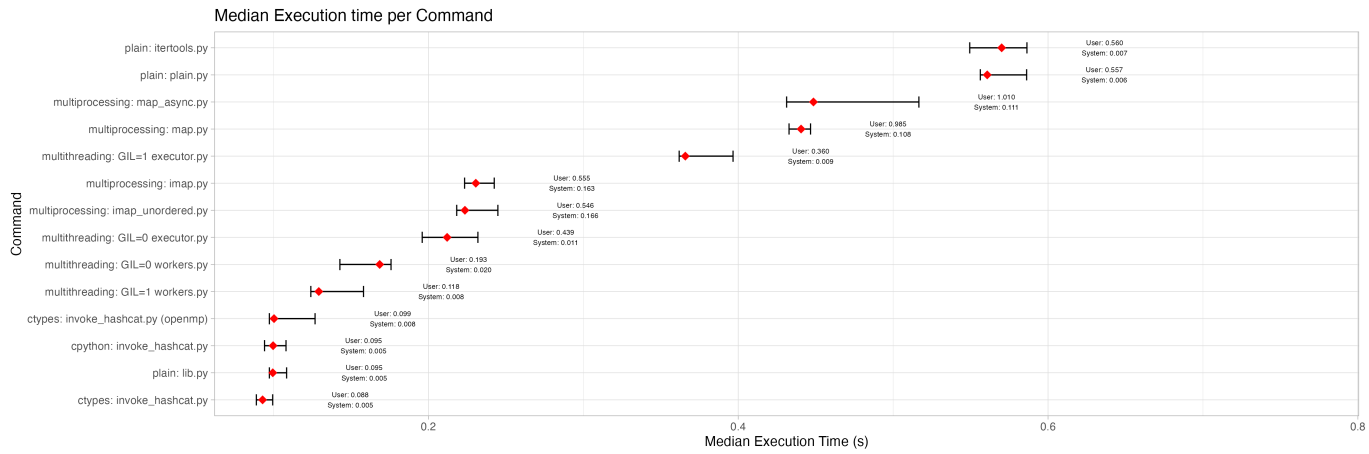

Figure 1: Performance Overview

[18] https://github.com/python/cpython/blob/main/Include/Python.h

command	mean	stddev	median	user	system	min	max
plain: itertools.py	0.5674692	0.0119883	0.5700655	0.5599028	0.0074184	0.5496380	0.5865690
plain: lib.py	0.1003698	0.0026018	0.0995592	0.0950988	0.0051505	0.0976092	0.1086287
plain: plain.py	0.5631182	0.0085463	0.5607433	0.5574100	0.0056163	0.5564152	0.5863659
multiprocessing: imap_unordered.py	0.2258019	0.0085966	0.2235880	0.5456730	0.1661910	0.2184310	0.2449692
multiprocessing: imap.py	0.2328316	0.0065632	0.2306093	0.5554529	0.1625106	0.2235672	0.2426373
multiprocessing: map_async.py	0.4528332	0.0248580	0.4485649	1.0100400	0.1108107	0.4314743	0.5167882
multiprocessing: map.py	0.4400746	0.0043315	0.4405771	0.9853628	0.1084339	0.4329375	0.4467715
multithreading: GIL=1 executor.py	0.3696592	0.0103798	0.3658508	0.3597924	0.0092238	0.3621005	0.3968615
multithreading: GIL=0 executor.py	0.2102704	0.0104267	0.2121045	0.4389796	0.0105094	0.1962787	0.2321854
multithreading: GIL=1 workers.py	0.1304648	0.0071726	0.1292655	0.1178397	0.0081775	0.1242261	0.1582329
multithreading: GIL=0 workers.py	0.1677349	0.0076839	0.1685633	0.1931002	0.0202853	0.1429964	0.1760283
ctypes: invoke_hashcat.py	0.0934947	0.0031416	0.0929726	0.0882496	0.0049164	0.0891827	0.0996272
ctypes: invoke_hashcat.py (openmp)	0.1021338	0.0056378	0.1003631	0.0986943	0.0083828	0.0976012	0.1269725
cpython: invoke_hashcat.py	0.1006056	0.0043579	0.0997623	0.0950439	0.0052310	0.0943297	0.1081794

Each experiment was conducted with 3 warmup runs using the `hyperfine` library. The following observations can be made:

- The fastest GIL-enabled parallelization using `imap_unordered` achieved a ~2.5x speedup over the plain python implementation.
- The fastest GIL-free parallelization implementation, individually managing threads, managed to shave off another ~0.06s, achieving a ~3.4x speedup over the plain python implementation.
- The overhead induced by OpenMP made it an unviable option for parallelization in our case.
- The ctypes implementation outperformed the extension of CPython directly and achieved a ~6x speedup over the plain python implementation, even marginally outperforming the usage of the `hashlib.sha1` library in plain Python.

Our findings, while insightful, are limited in their generalizability due to the simplicity of the experiments, the overhead introduced by containerization and various implicit assumptions about the underlying hardware and software environment. Nonetheless, this report serves as both a proof of concept and a practical guide for engineering parallelized Python code across different abstraction levels. It highlights the trade-offs between performance, complexity and usability.

Crucially, we demonstrate that significant performance gains can be achieved by leveraging Python's robust ecosystem and its efficient bindings to C libraries. For instance, a mere four lines of Python code using the `hashlib.sha1` library outperformed several of our more intricate C implementations. This underscores the remarkable efficiency of Python's ecosystem and the importance of focusing on high-level optimization strategies—leveraging well-optimized libraries rather than reinventing the wheel.

Addendum

All experiments were conducted on a consumer-grade laptop with the following specifications:

```
$ system_profiler SPSoftwareDataType SPHardwareDataType

Software:

    System Software Overview:

      System Version: macOS 14.6.1 (23G93)
      Kernel Version: Darwin 23.6.0
      Boot Volume: Macintosh HD
      Boot Mode: Normal
      Computer Name: Yahya's MacBook Pro
      User Name: Yahya Jabary (sueszli)
      Secure Virtual Memory: Enabled
      System Integrity Protection: Enabled
      Time since boot: 79 days, 22 hours, 26 minutes

Hardware:

    Hardware Overview:
```

```
      Model Name: MacBook Pro
      Model Identifier: Mac14,10
      Model Number: <redacted>
      Chip: Apple M2 Pro
      Total Number of Cores: 12 (8 performance and 4 efficiency)
      Memory: 16 GB
```

```
$ docker compose exec main lscpu

Architecture:                         x86_64
CPU op-mode(s):                       32-bit
Byte Order:                           Little Endian
CPU(s):                               12
On-line CPU(s) list:                  0-11
Thread(s) per core:                   1
Core(s) per socket:                   12
Socket(s):                            1
Vendor ID:                            0x61
Model:                                0
Stepping:                             0x0
BogoMIPS:                             48.00
Vulnerability Gather data sampling:   Not affected
Vulnerability Itlb multihit:          Not affected
Vulnerability L1tf:                   Not affected
Vulnerability Mds:                    Not affected
Vulnerability Meltdown:               Not affected
Vulnerability Mmio stale data:        Not affected
Vulnerability Reg file data sampling: Not affected
Vulnerability Retbleed:               Not affected
Vulnerability Spec rstack overflow:   Not affected
Vulnerability Spec store bypass:      Mitigation; Speculative Store Bypass disabled via prctl
Vulnerability Spectre v1:             Mitigation; __user pointer sanitization
Vulnerability Spectre v2:             Not affected
Vulnerability Srbds:                  Not affected
Vulnerability Tsx async abort:        Not affected
Flags:                                fp asimd evtstrm aes pmull sha1 sha2 crc32 atomics fphp asimdhp cpuid asimdrdm jscvt fcma lrc
                                       ilrcpc flagm ssbs sb paca pacg dcpodp flagm2 frint
```