Breaking free from the GIL

Code: `github.com/sueszli/nogil` Yahya Jabary, TU Wien

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.

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] 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

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.

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.
- 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.).

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

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.

Results

We beat `hashlib` by 13.525ns (2.5x) or 101,703,681 instructions (3.3x). This was achieved through the `ctypes` library, CPython-C-API and various C libraries.

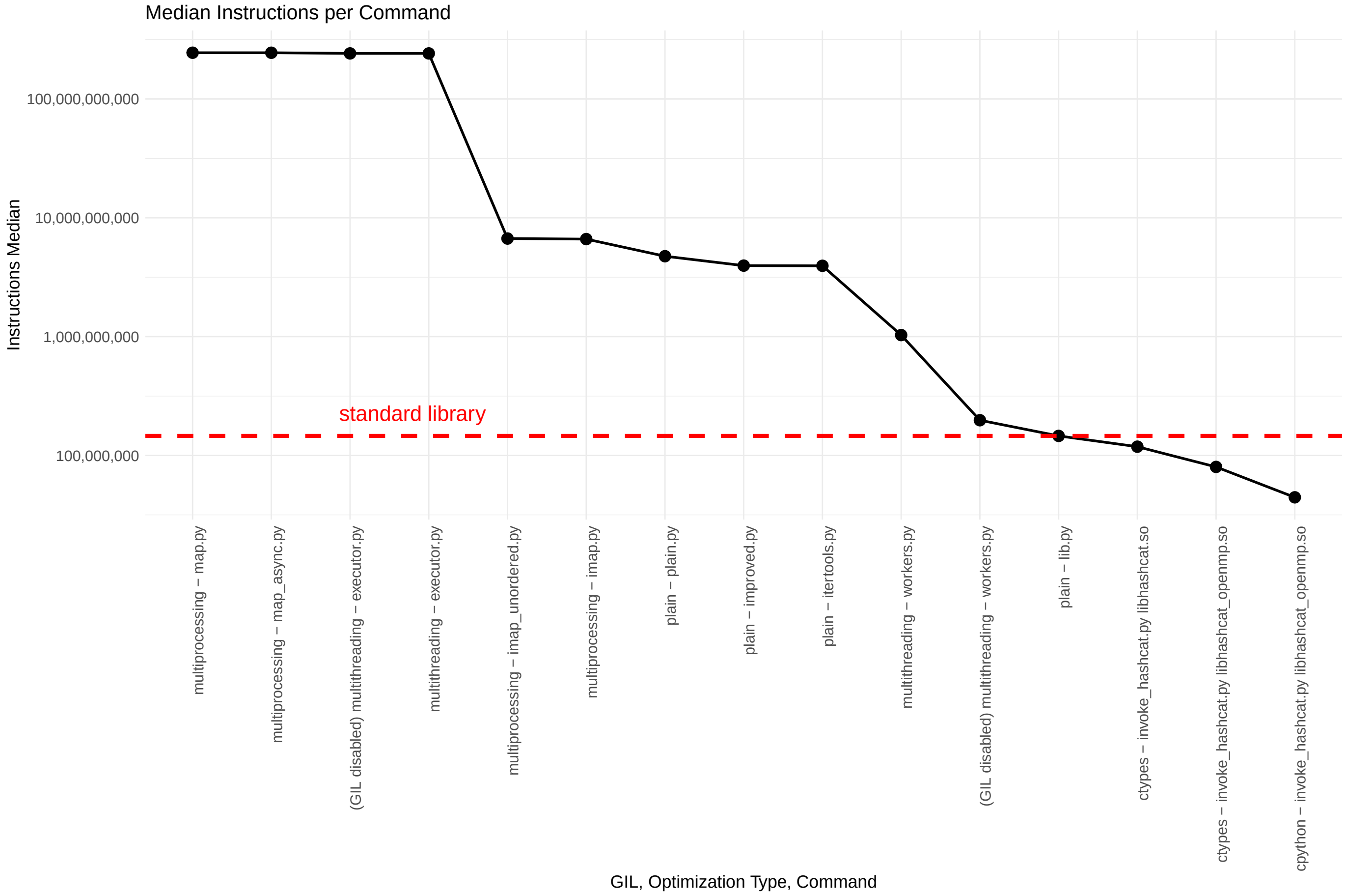
gil	type	command	instructions (med)	task_clock (med)	user_time (med)	sys_time (med)
true	**cpython**	**invoke_hashcat.py (openmp)**	44442376	8.760	0.0090265	0.0000000
true	ctypes	invoke_hashcat.py (openmp)	80107280	39.820	0.0160960	0.0000000
true	ctypes	invoke_hashcat.py	118592997	16.110	0.0162195	0.0000000
true	**plain**	**lib.py (hashlib libary)**	146146058	22.285	0.0222055	0.0000000
false	multithreading	workers.py	198008716	39.985	0.0258195	0.0113530
true	multithreading	workers.py	1030919157	106.765	0.1006565	0.0111120
true	plain	itertools.py	3945750392	325.575	0.3242800	0.0000000
true	plain	improved.py	3959326962	322.015	0.3206755	0.0000000
true	plain	plain.py	4752510454	400.205	0.3996415	0.0000000
true	multiprocessing	imap.py	6620723294	1743.685	1.2987810	0.4785180
true	multiprocessing	imap_unordered.py	6692752894	1787.350	1.3024570	0.5340625
true	multithreading	executor.py	241741072306	20136.600	19.8526395	0.6276710
false	multithreading	executor.py	241749347062	63354.890	63.2586825	0.0327220
true	multiprocessing	map_async.py	244913585430	61218.555	61.0370955	0.2066270
true	multiprocessing	map.py	245013383854	61259.295	61.0844710	0.2048880

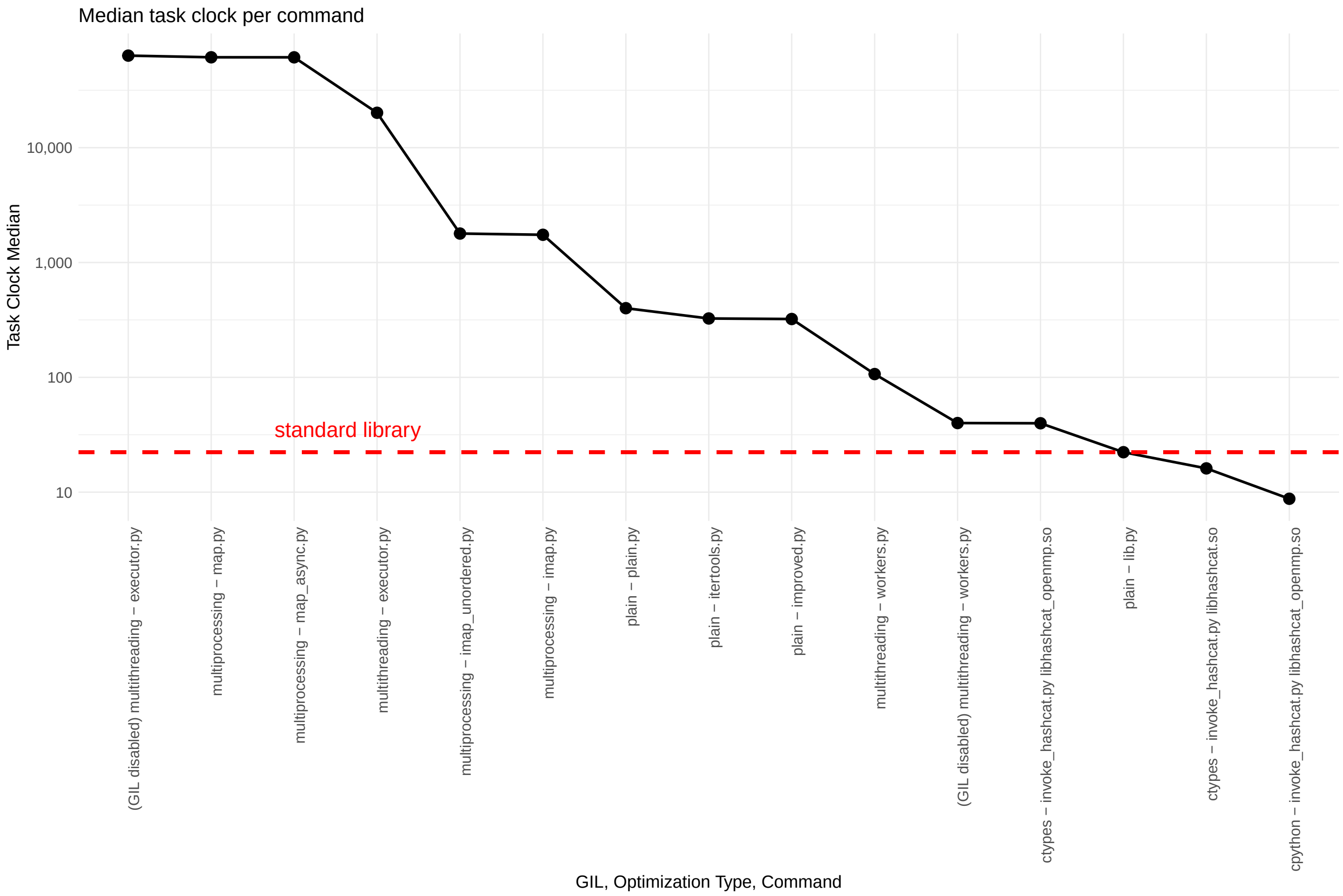
More importantly, disabling the GIL while individually managing threads with the `multithreading` API reduced time spent on syscalls by 42x. This shows that the GIL is a significant bottleneck for parallel computing in Python. Nonetheless, the single threaded implementation invoking the `hashlib` library, our `ctypes` implementation or the CPython extension are most likely the best choice for most performance critical applications.

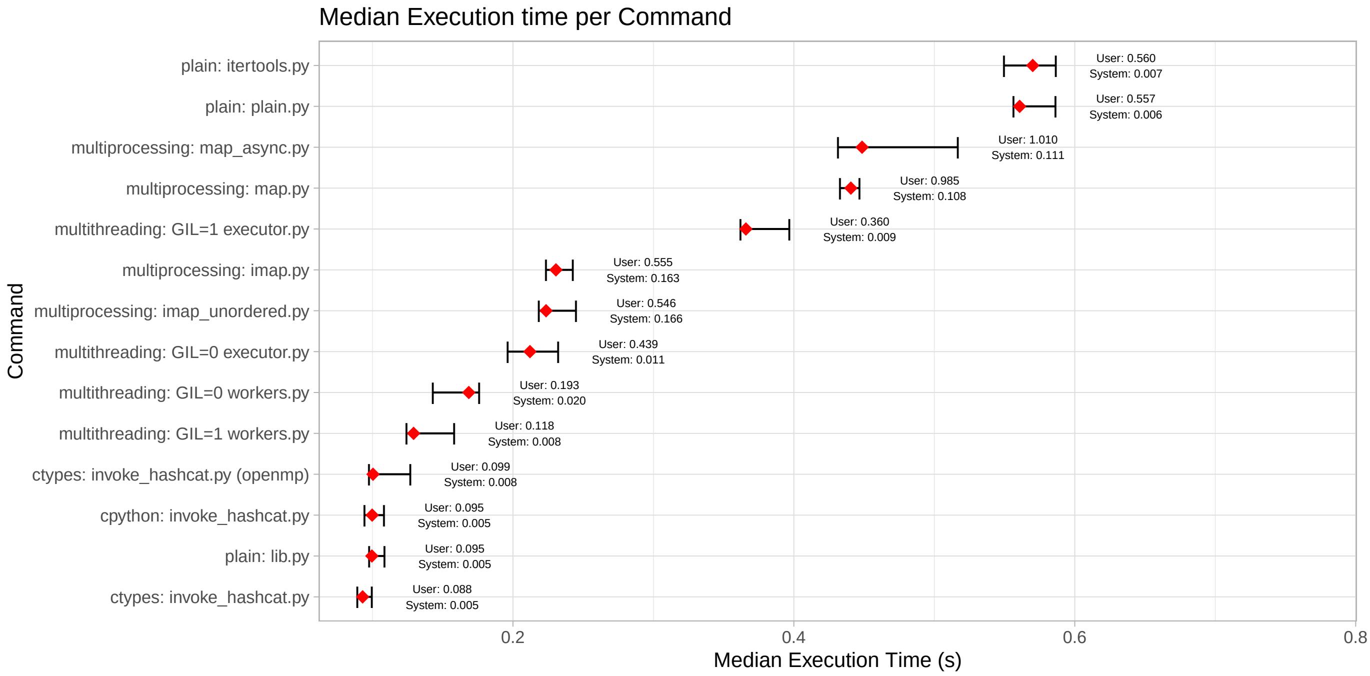
[18] https://github.com/python/cpython/blob/main/Include/Python.h

All experiments were conducted on an 11th Gen Intel Core i7-1165G7 processor running at 2.80 GHz with 16 GB of RAM.

These insights provide a valuable foundation for future work in optimizing Python performance and offer practical guidance for developers seeking to enhance the efficiency of their applications.







Median Execution time per Command
Command
plain: itertools.py
plain: plain.py
multiprocessing: map_async.py
multiprocessing: map.py
multithreading: GIL=1 executor.py
multiprocessing: imap.py
multiprocessing: imap_unordered.py
multithreading: GIL=0 executor.py
multithreading: GIL=0 workers.py
multithreading: GIL=1 workers.py
ctypes: invoke_hashcat.py (openmp)
cpython: invoke_hashcat.py
plain: lib.py
ctypes: invoke_hashcat.py
User: 0.560
System: 0.007
User: 0.557
System: 0.006
User: 1.010
System: 0.111
User: 0.985
System: 0.108
User: 0.360
System: 0.009
User: 0.555
System: 0.163
User: 0.546
System: 0.166
User: 0.439
System: 0.011
User: 0.193
System: 0.020
User: 0.118
System: 0.008
User: 0.099
System: 0.008
User: 0.095
System: 0.005
User: 0.095
System: 0.005
User: 0.088
System: 0.005
0.2
0.4
0.6
0.8
Median Execution Time (s)