Breaking free from the GIL

Group 07: Efficient Programs, Prof. Anton Ertl

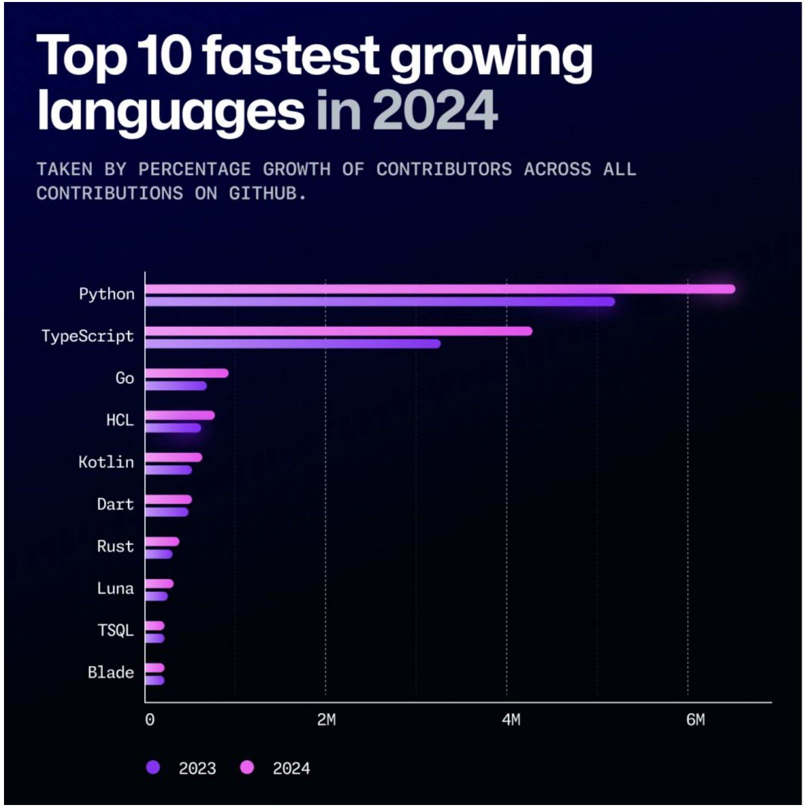
Code: github.com/sueszli/nogil

Report: sueszli.github.io/nogil/docs/report.pdf

Why Python?

- most popular since oct 24
- simple and “pythonic”
- garbage-collected
- dynamically-typed

- scripting
- data modeling
- scientific computing



Shortcomings

- `asyncio` is great for I/O-bound tasks
- GIL is bad for compute-bound tasks
  - GIL = global interpreter lock
  - mutex for bytecode

Shortcomings

- `asyncio` is great for I/O-bound tasks
- GIL is bad for compute-bound tasks
    - GIL = global interpreter lock
    - mutex for bytecode

"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."

- Rob Pike 2012

Workarounds

- super-languages (mojo, taichi)

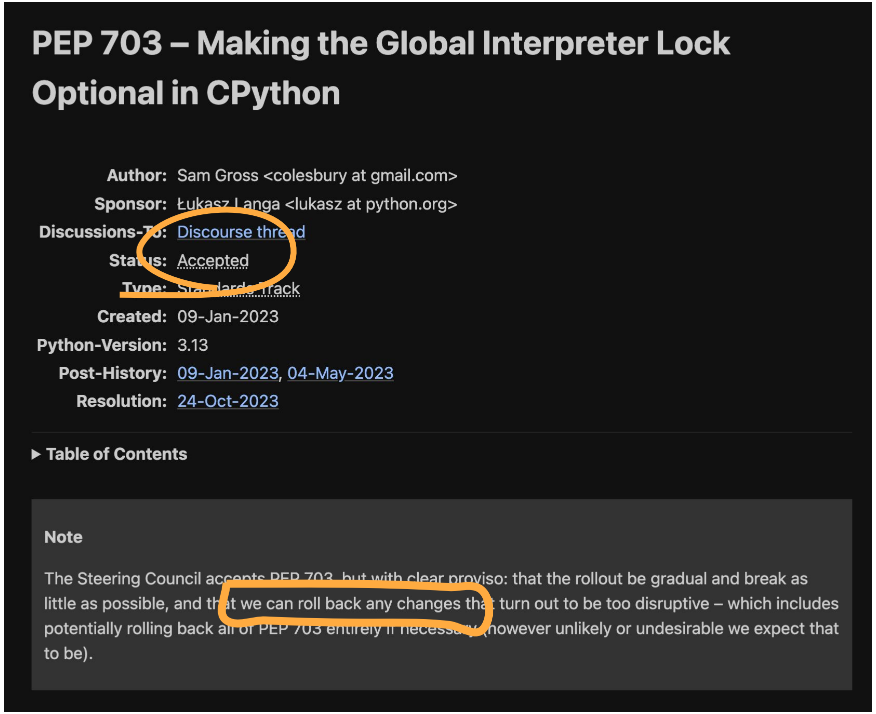
- JIT interpreters (pypy, numba)

- lightweight sub-interpreters (PEP 554)

- optional GIL (PEP 703)

    - previously only through C-interop
    - now also in vanilla python!

- devs are scared of breaking backwards compatibility



PEP 703 – Making the Global Interpreter Lock Optional in CPython

Author: Sam Gross <colesbury at gmail.com>
Sponsor: Łukasz Langa <lukasz at python.org>
Discussions-To: Discourse thread
Status: Accepted
Type: Standards Track
Created: 09-Jan-2023
Python-Version: 3.13
Post-History: 09-Jan-2023, 04-May-2023
Resolution: 24-Oct-2023

▶ **Table of Contents**

Note

The Steering Council accepts PEP 703, but with clear proviso: that the rollout be gradual and break as little as possible, and that we can roll back any changes that turn out to be too disruptive – which includes potentially rolling back all of PEP 703 entirely if necessary (however unlikely or undesirable we expect that to be).

Experiments

Algorithm: collision attack

hashcat
advanced password recovery

find value *x* that was passed to *hash(x)*.

- naive brute force, breadth first search.
- embarrassingly parallel.

implemented from scratch:

- sha256: 7870.21it/s
- md5: 9847.03it/s
- sha1: 18578.26it/s (insecure, but fast enough for eval)

Optimization Strategies

1. plain python
2. multiprocessing
3. multithreading
4. ctypes
5. cpython

Optimization Strategies

1. plain python
    - vanilla python + `hashlib` (baseline)
    - optimize with loop unrolling, method inlining
2. multiprocessing
3. multithreading
    - also disabling the GIL
4. ctypes
5. cpython
    - extending the cPython interpreter (`Python.h`)

Plain Python

```python
def hashcat(target_hash, max_length=8):
    alphabet = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
    position = [0] * max_length

    for length in range(1, max_length + 1):
        while True:
            current = "".join(alphabet[position[i]] for i in range(length))
            hashed = sha1(current).hex()
            if hashed == target_hash:
                return current

            idx = 0
            while idx < length:
                position[idx] += 1
                if position[idx] < len(alphabet):
                    break
                position[idx] = 0
                idx += 1

            if idx == length:
                break

    return None
```

Plain Python: Loop unrolling, method inlining

```python
def sha1(msg):
    if isinstance(msg, str):
        msg = msg.encode()
    assert isinstance(msg, bytes)

    ml = len(msg) * 8
    msg += b"\x80"
    msg += b"\x00" * (-(len(msg) + 8) % 64)
    msg += bytes([(ml >> (56 - i * 8)) & 0xFF for i in range(8)])

    width = 32
    lrot = lambda value, n: ((value << n) & 0xFFFFFFFF) | (value >> (width - n))
    bytes_to_word = lambda b: (b[0] << 24) | (b[1] << 16) | (b[2] << 8) | b[3]

    h = [0x67452301, 0xEFCDAB89, 0x98BADCFE, 0x10325476, 0xC3D2E1F0]
    for chunk in [msg[i : i + 64] for i in range(0, len(msg), 64)]:
        w = [bytes_to_word(chunk[i : i + 4]) for i in range(0, 64, 4)]

        for i in range(16, 80):
            w.append(lrot(w[i - 3] ^ w[i - 8] ^ w[i - 14] ^ w[i - 16], 1))
```

```python
def sha1(msg):
    if isinstance(msg, str):
        msg = msg.encode()
    assert isinstance(msg, bytes)

    ml = len(msg) * 8
    msg += b"\x80"
    msg += b"\x00" * (-(len(msg) + 8) % 64)
    msg += bytes([(ml >> (56 - i * 8)) & 0xFF for i in range(8)])

    h = [0x67452301, 0xEFCDAB89, 0x98BADCFE, 0x10325476, 0xC3D2E1F0]
    K = [0x5A827999, 0x6ED9EBA1, 0x8F1BBCDC, 0xCA62C1D6]

    for i in range(0, len(msg), 64):
        chunk = msg[i:i + 64]
        w = [
            (chunk[j] << 24) | (chunk[j + 1] << 16) | (chunk[j + 2] << 8) | chunk[j + 3]
            for j in range(0, 64, 4)
        ]

        for j in range(16, 80):
            value = w[j - 3] ^ w[j - 8] ^ w[j - 14] ^ w[j - 16]
            w.append(((value << 1) & 0xFFFFFFFF) | (value >> 31))
```

Plain Python: Loop unrolling, method inlining

```python
        a, b, c, d, e = h
        for i in range(len(w)):
            if i < 20:
                f, k = d ^ (b & (c ^ d)), 0x5A827999
            elif i < 40:
                f, k = b ^ c ^ d, 0x6ED9EBA1
            elif i < 60:
                f, k = (b & c) | (d & (b | c)), 0x8F1BBCDC
            else:
                f, k = b ^ c ^ d, 0xCA62C1D6
            tmp = (lrot(a, 5) + f + e + k + w[i]) & 0xFFFFFFFF
            e, d, c, b, a = d, c, lrot(b, 30), a, tmp
        h = [((v + n) & 0xFFFFFFFF) for v, n in zip(h, [a, b, c, d, e])]

    return b"".join([v.to_bytes(4, "big") for v in h])
```

```python
        a, b, c, d, e = h
        for j in range(20):
            f = d ^ (b & (c ^ d))
            tmp = (((a << 5) & 0xFFFFFFFF) | (a >> 27)) + f + e + K[0] + w[j]
            e, d, c, b, a = d, c, ((b << 30) & 0xFFFFFFFF) | (b >> 2), a, tmp & 0xFFFFFFFF
        for j in range(20, 40):
            f = b ^ c ^ d
            tmp = (((a << 5) & 0xFFFFFFFF) | (a >> 27)) + f + e + K[1] + w[j]
            e, d, c, b, a = d, c, ((b << 30) & 0xFFFFFFFF) | (b >> 2), a, tmp & 0xFFFFFFFF
        for j in range(40, 60):
            f = (b & c) | (d & (b | c))
            tmp = (((a << 5) & 0xFFFFFFFF) | (a >> 27)) + f + e + K[2] + w[j]
            e, d, c, b, a = d, c, ((b << 30) & 0xFFFFFFFF) | (b >> 2), a, tmp & 0xFFFFFFFF
        for j in range(60, 80):
            f = b ^ c ^ d
            tmp = (((a << 5) & 0xFFFFFFFF) | (a >> 27)) + f + e + K[3] + w[j]
            e, d, c, b, a = d, c, ((b << 30) & 0xFFFFFFFF) | (b >> 2), a, tmp & 0xFFFFFFFF

        h[0] = (h[0] + a) & 0xFFFFFFFF
        h[1] = (h[1] + b) & 0xFFFFFFFF
        h[2] = (h[2] + c) & 0xFFFFFFFF
        h[3] = (h[3] + d) & 0xFFFFFFFF
        h[4] = (h[4] + e) & 0xFFFFFFFF

    return b"".join(v.to_bytes(4, "big") for v in h)
```

Optimization Strategies

1. plain python
    - vanilla python + `hashlib` (baseline)
    - optimize with loop unrolling, method inlining
2. multiprocessing
3. multithreading
    - also disabling the GIL
4. ctypes
5. cpython
    - extending the cPython interpreter (`Python.h`)

Optimization Strategies

1. plain python
   - vanilla python + `hashlib` (baseline)
   - optimize with loop unrolling, method inlining
2. multiprocessing
3. multithreading
   - also disabling the GIL
4. ctypes
5. cpython
   - extending the cPython interpreter (`Python.h`)

- compile v3.13
- use `PYTHON_GIL=0` flag
- try a bunch of functions

Multithreading: Barrier pattern

very similar to C equivalent

```python
def hashcat(target_hash, max_length=8):
    import os
    import string
    from itertools import product
    from queue import Queue
    from threading import Event, Thread

    alphabet = string.ascii_letters + string.digits
    work_queue = Queue()
    result_queue = Queue()
    found_event = Event()
    num_threads = os.cpu_count() * 2

    threads = []
    for _ in range(num_threads):
        t = Thread(target=worker, args=(work_queue, target_hash, found_event, result_queue))
        t.daemon = True
        t.start()
        threads.append(t)

    for length in range(1, max_length + 1):
        if found_event.is_set():
            break

        def chunk_generator(iterable, chunk_size=1000):
            chunk = []
            for item in iterable:
                chunk.append(item)
                if len(chunk) == chunk_size:
                    yield chunk
                    chunk = []
            if chunk:
                yield chunk

        guesses = ("".join(guess) for guess in product(alphabet, repeat=length))
        for chunk in chunk_generator(guesses):
            work_queue.put(chunk)
            if found_event.is_set():
                break

    # barrier
    for _ in threads:
        work_queue.put(None)
    for t in threads:
        t.join()

    # check if any success
    if not result_queue.empty():
        return result_queue.get()
    return None


if __name__ == "__main__":
    import sys

    assert len(sys.argv) == 2
    target_hash = sys.argv[1]
    password = hashcat(target_hash)
```

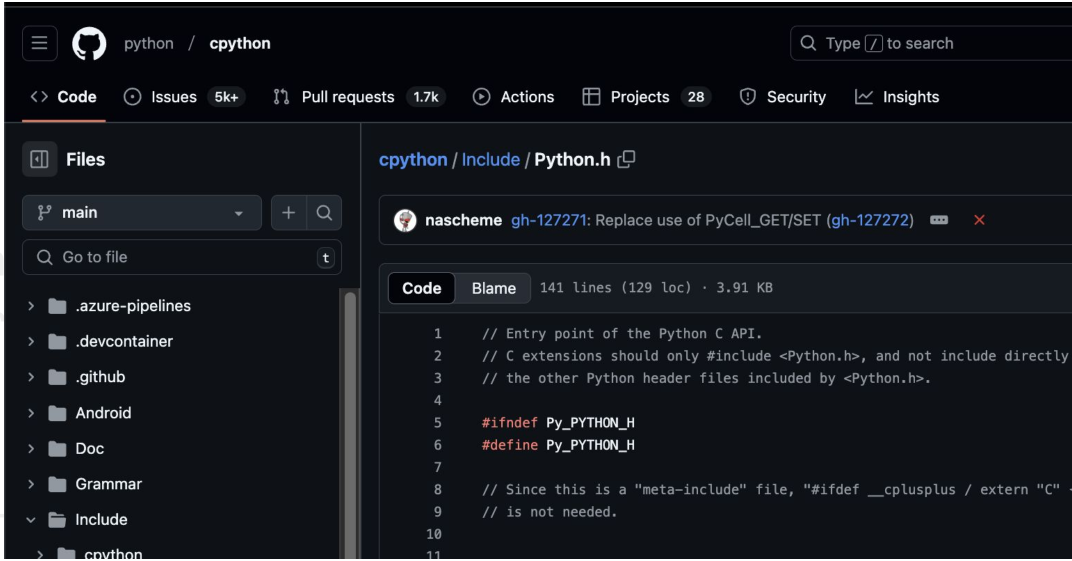
Optimization Strategies

1. plain python
   - vanilla python + `has
   - optimize with loop un
2. multiprocessing
3. multithreading
   - also disabling the GI
4. ctypes
5. cpython
   - extending the cPython

```python
def hashcat(target_hash, shared_lib):
    import ctypes

    lib = ctypes.CDLL(shared_lib)

    # `char* hashcat(const char *target_hash)`
    lib.hashcat.argtypes = [ctypes.c_char_p]
    lib.hashcat.restype = ctypes.c_char_p

    hash_bytes = target_hash.encode("utf-8")
    result = lib.hashcat(hash_bytes)
    return result.decode("utf-8")
```

Optimization Strategies

1. plain python
    - vanilla python + `hash
    - optimize with loop unr
2. multiprocessing
3. multithreading
    - also disabling the GIL
4. ctypes
5. cpython
    - extending the cPython interpreter (`Python.h`)



Trade Offs

Optimization Strategies

1. plain python
2. multiprocessing
3. multithreading
4. ctypes
5. cpython

Multiprocessing

- **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.
- **Serialization overhead:** there is no shared memory, so data has to be serialized and deserialized for inter-process communication. Also, some objects are unserializable / not pickleable (i.e. lambdas, file handles, …).

Multiprocessing *vs. Multithreading*

- **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.
- **Serialization overhead:** there is no shared memory, so data has to be serialized and deserialized for inter-process communication. Also, some objects are unserializable / not pickleable (i.e. lambdas, file handles, …).

Ctypes

- a lot simpler than cpython extensions
- foreign function interface (FFI) for Python that allows calling functions from shared libraries
- extremely high serialization overhead (but passing pointers is possible)
- not meant for HPC but codebase glue

CPython Extensions

- bare metal, zero overhead

- `mmap()` allows sharing huge chunks of memory

- very complex API, requires you to manually manage the GIL with `Py_BEGIN_ALLOW_THREADS` and `Py_END_ALLOW_THREADS` macros and marshal all data passed.

- not portable, requires a compile step.

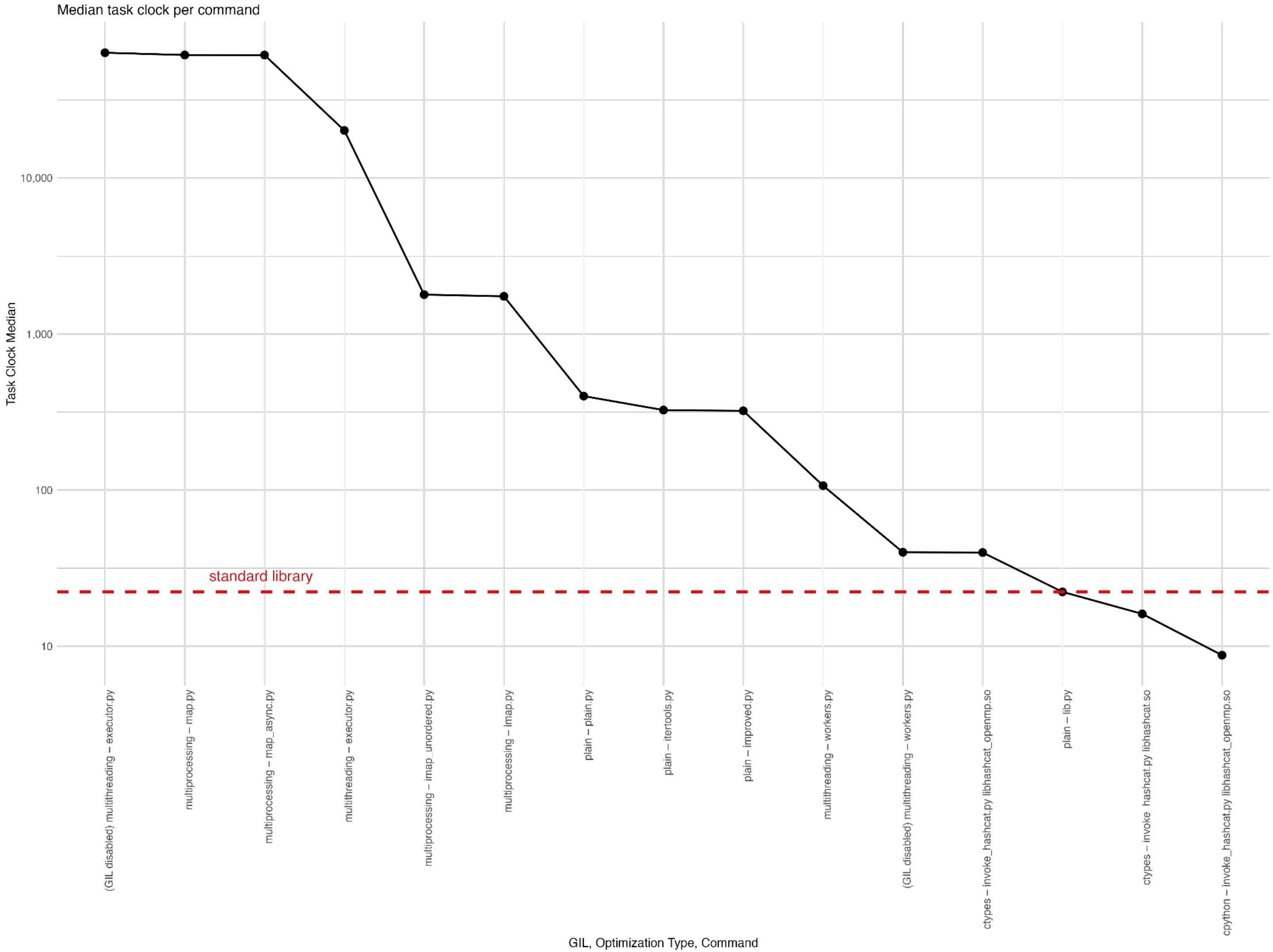
Final Results

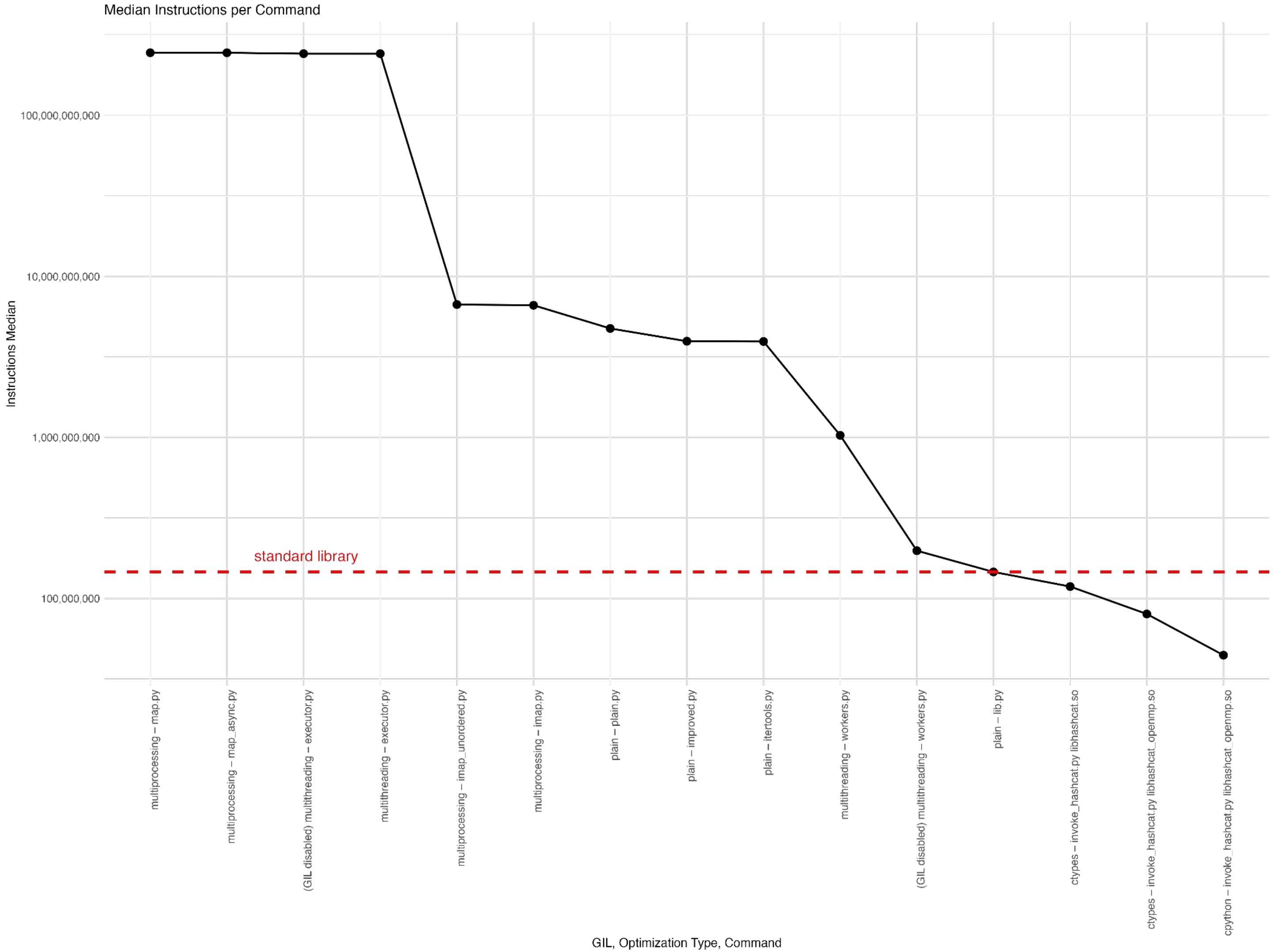
Evaluation

- `perf` unix tool
- `hyperfine` rust library

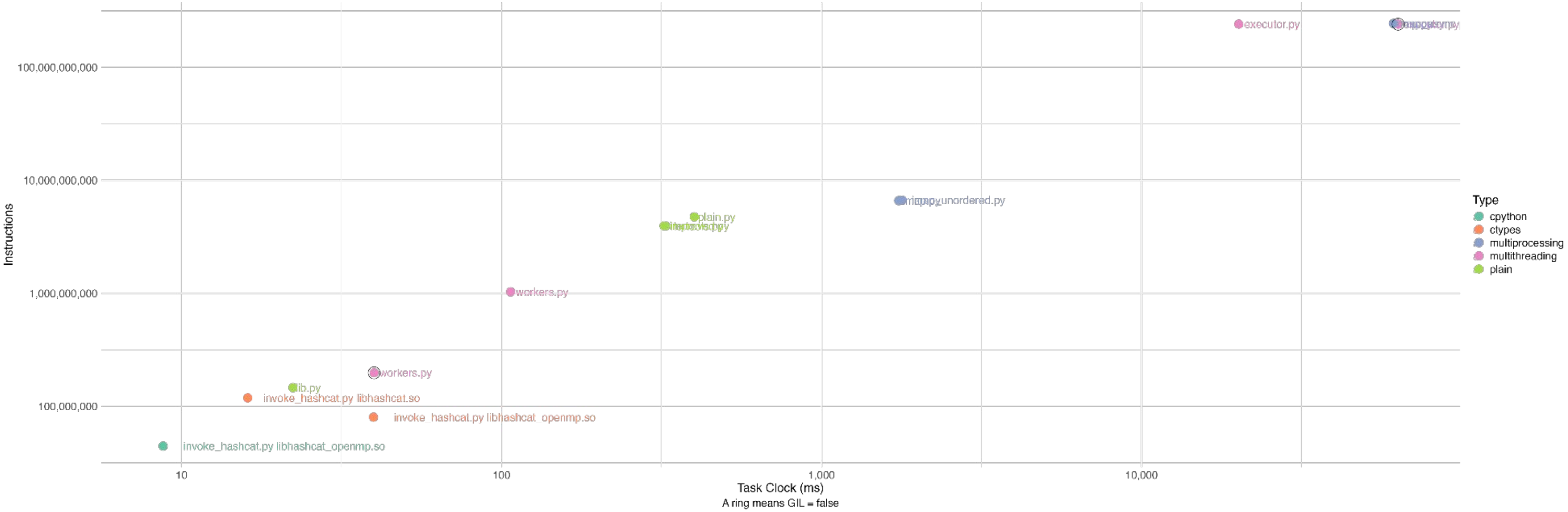
We beat `hashlib` by 13.525ns (2.5x) or 101,703,681 instructions (3.3x). This was achieved through the `ctypes` library, CPython-C-API and various C libraries.

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


Median task clock per command
Task Clock Median
10,000
1,000
100
10
standard library
(GIL disabled) multithreading – executor.py
multiprocessing – map.py
multiprocessing – map_async.py
multithreading – executor.py
multiprocessing – imap_unordered.py
multiprocessing – imap.py
plain – plain.py
plain – itertools.py
plain – improved.py
multithreading – workers.py
(GIL disabled) multithreading – workers.py
ctypes – invoke_hashcat_openmp.py libhashcat_openmp.so
plain – lib.py
ctypes – invoke_hashcat.py libhashcat.so
cpython – invoke_hashcat_openmp.py libhashcat_openmp.so
GIL, Optimization Type, Command



Median Instructions per Command
Instructions Median
100,000,000,000
10,000,000,000
1,000,000,000
100,000,000
standard library
multiprocessing – map.py
multiprocessing – map_async.py
(GIL disabled) multithreading – executor.py
multithreading – executor.py
multiprocessing – imap_unordered.py
multiprocessing – imap.py
plain – plain.py
plain – improved.py
plain – itertools.py
multithreading – workers.py
(GIL disabled) multithreading – workers.py
plain – lib.py
ctypes – invoke_hashcat.py libhashcat.so
ctypes – invoke_hashcat.py libhashcat_openmp.so
cpython – invoke_hashcat.py libhashcat_openmp.so
GIL, Optimization Type, Command



Instructions
100,000,000,000
10,000,000,000
1,000,000,000
100,000,000
Task Clock (ms)
10
100
1,000
10,000
A ring means GIL = false
Type
cpython
ctypes
multiprocessing
multithreading
plain
executor.py
imap_unordered.py
plain.py
workers.py
lib.py
invoke_hashcat.py libhashcat.so
invoke_hashcat.py libhashcat_openmp.so

Thanks!