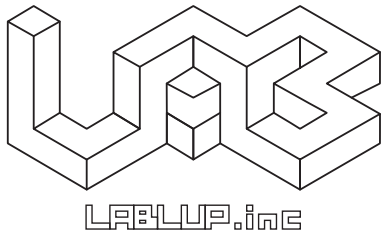
High-Performance Networking in Python

Joongi Kim (김준기)

Lablup Inc. / @achimnol

2016. 8. 13

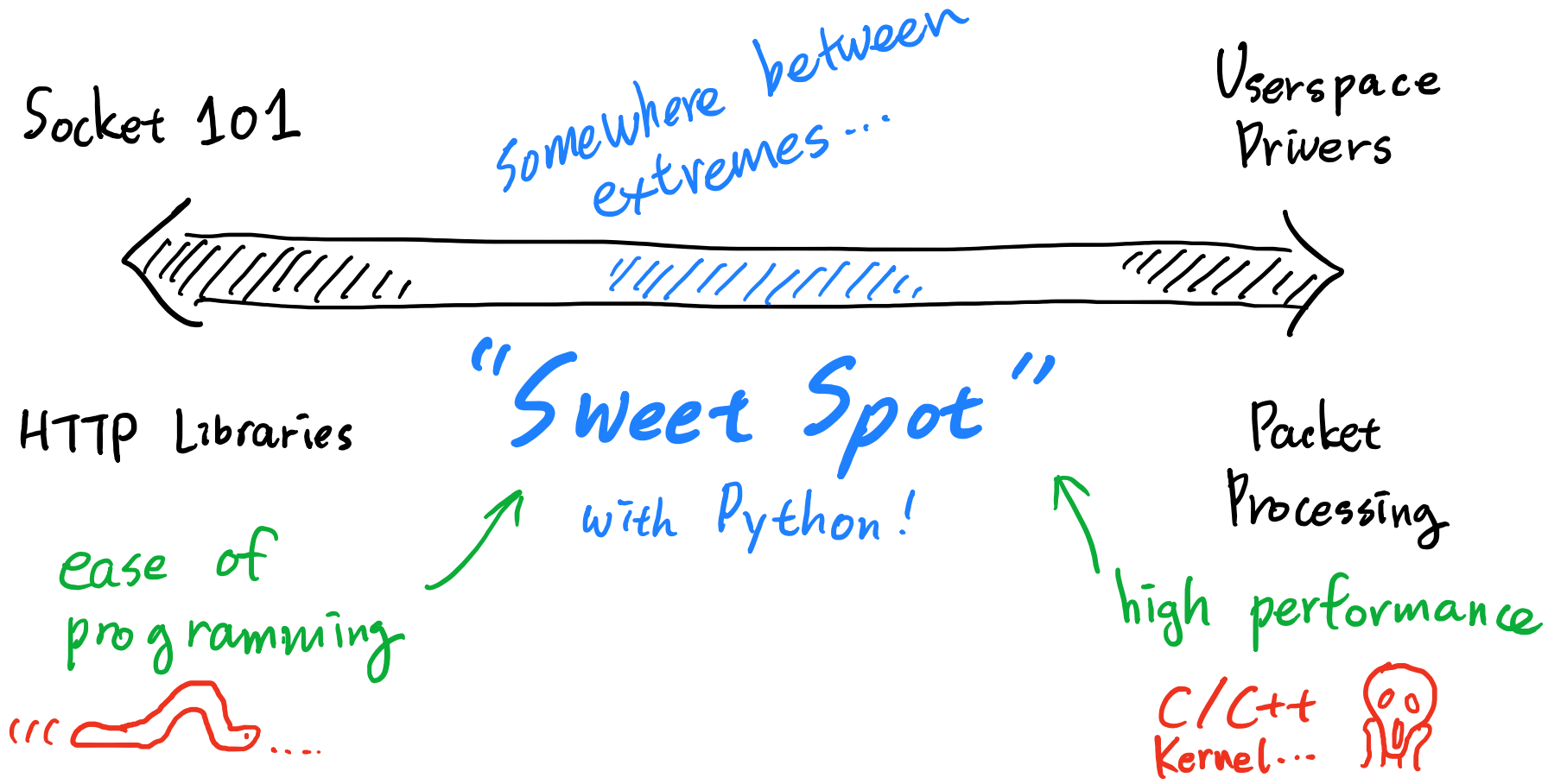


My Background

- Ph.D in Computer Science at KAIST
  - Designed a packet processing framework using heterogeneous processors (CUDA GPUs + Intel Xeon Phi)
    - 80 Gbps on a single x86 Linux server (Intel DPDK + NBA framework)
    - about 48K lines of C++
    - https://github.com/anlab-kaist/NBA
- CTO at Lablup Inc.
  - Developing a distributed sand-boxed code execution service
    - Python + ZeroMQ + AsyncIO
    - http://www.lablup.com

Motivation and Goal

- To let you grasp key principles for high-performance networking
- To introduce modern Python networking schemes



Technical Background

- I assume that you have...
  - Knowledge of socket programming
  - Basic experience on building server applications (e.g., echo server)
  - Understandings of network stack (e.g., TCP / IP)
  - Familiarity with Python standard library
  - Understandings of multi-threading in operating systems

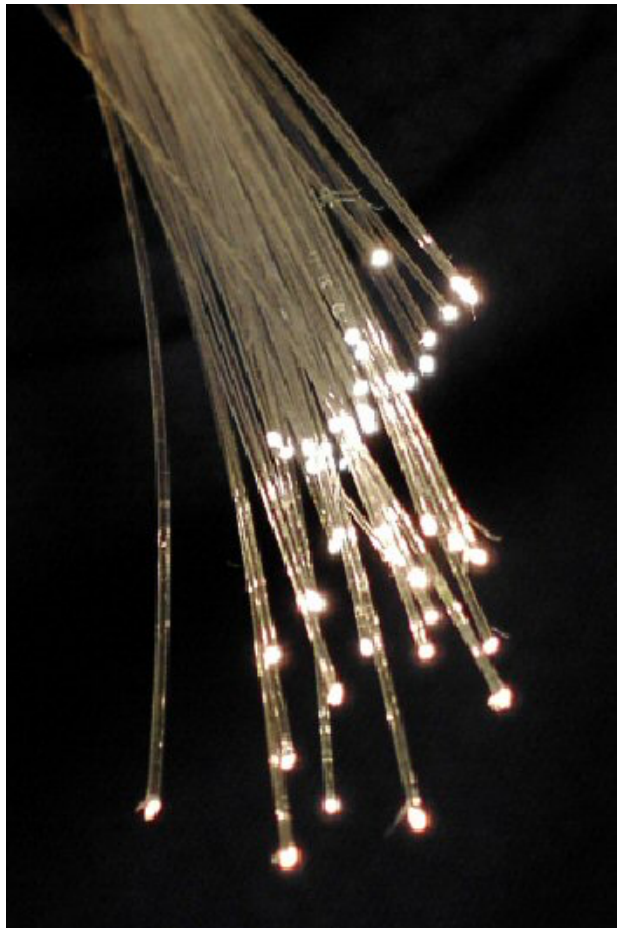
- Python version for this talk: 3.5.2+

Contents

- Multiplexing I/O in networking
- Complexity of manual event loop implementation
- Generators, Coroutines, and Python asyncio
- Tips for learning and using asyncio
- Achieving high-performance with asyncio
- Alternative approach: PyParallel
- Closing

Fundamental Issues in Networking

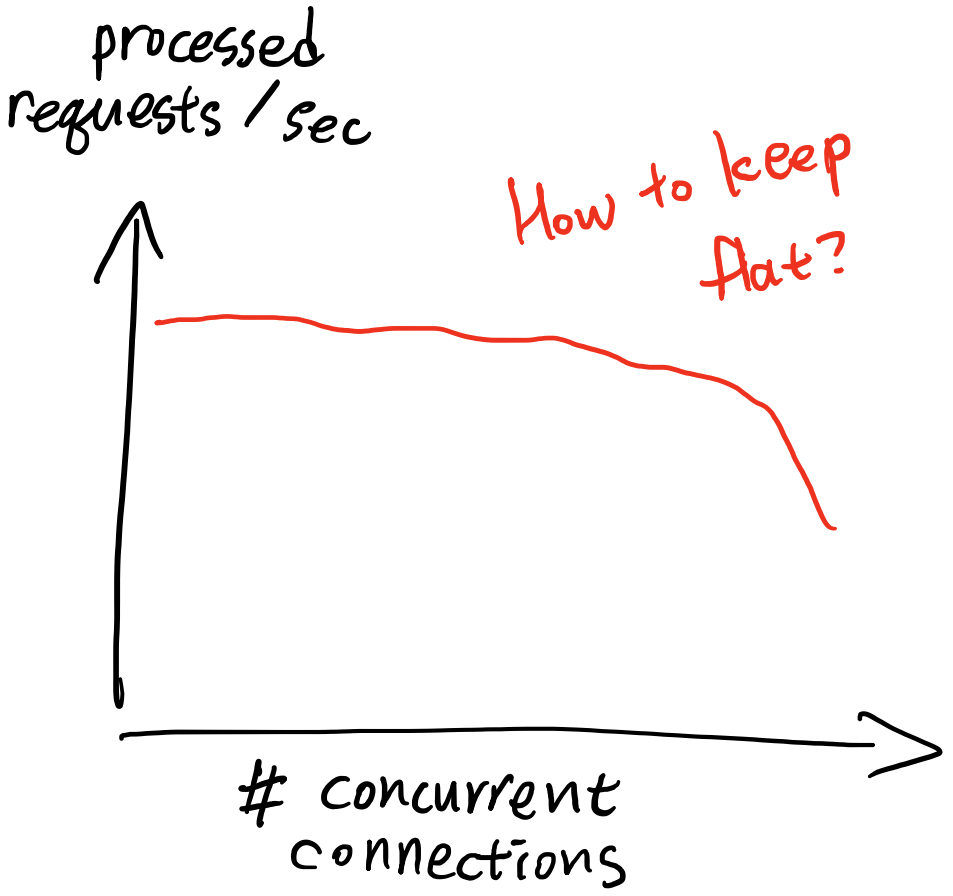
- Goal : communication with many peers
- What do we need?
  - Reliable data communication
  - *Multiplexing* multiple communication channels
- Who is responsible for multiplexing?
  - Operating systems
  - Programming languages & runtimes
  - You?!



The C10K Problem

- http://www.kegel.com/c10k.html
  - Goal : 10K clients served by a single server

- Why difficult?
  - *"This has not yet become popular in Unix, probably because few operating systems support asynchronous I/O, also possibly because it (like non-blocking I/O) requires rethinking your application."*



Multiplexing Network Connections

	`fork` / `pthread_create`	`select` / `poll` / `epoll` / `kqueue`
Method	Create a new parallel context for every new connection	Get “ready to use” file descriptors from a set of file descriptors to monitor
Advantages	Simple to write programs using blocking calls	Less performance overheads (depending on underlying kernel implementation)
Disadvantages	Context switching overheads & High memory consumption	Difficult to write programs due to manual context tracking

What Happens If You Write Event Loop?

```
import selectors, socket

sel = selectors.DefaultSelector()

def accept(sock, mask):
  conn, addr = sock.accept()
  conn.setblocking(False)
  sel.register(conn,
               selectors.EVENT_READ,
               read)

def read(conn, mask):
  data = conn.recv(1024)
  if data:
    conn.send(data)
  else:
    sel.unregister(conn)
    conn.close()
```

```
# main program
sock = socket.socket()
sock.bind(('localhost', 1234))
sock.listen()
sock.setblocking(False)
sel.register(sock,
             selectors.EVENT_READ,
             accept)

while True:
  events = sel.select()
  for key, mask in events:
    callback = key.data
    callback(key.fileobj, mask)
```

What Happens If You Write Event Loop?

```
import selectors, socket

sel = selectors.DefaultSelector()

def accept(sock, mask):
  conn, addr = sock.accept()
  conn.setblocking(False)
  sel.register(conn,
               selectors.EVENT_READ,
               read)

def read(conn, mask):
  data = conn.recv(1024)
  if data:
    conn.send(data)
  else:
    sel.unregister(conn)
    conn.close()
```

What if not all 1024 bytes are recieved??

```
# main program
sock = socket.socket()
sock.bind(('localhost', 1234))
sock.listen()
sock.setblocking(False)
sel.register(sock,
             selectors.EVENT_READ,
             accept)

while True:
  events = sel.select()
  for key, mask in events:
    callback = key.data
    callback(key.fileobj, mask)
```

What Happens If You Write Event Loop?

```
import selectors, socket

sel = selectors.DefaultSelector()

def accept(sock, mask):
  conn, addr = sock.accept()
  conn.setblocking(False)
  sel.register(conn,
               selectors.EVENT_READ,
               read)

def read(conn, mask):
  data = conn.recv(1024)
  if data:
    conn.send(data)
  else:
    sel.unregister(conn)
    conn.close()
```

```
# main program
sock = socket.socket()
sock.bind(('localhost', 1234))
sock.listen()
sock.setblocking(False)
sel.register(sock,
             selectors.EVENT_READ,
             accept)

while True:
  events = sel.select()
  for key, mask in events:
    callback = key.data
```

What if not all 1024 bytes are recieved??

For sequential & blocking cases...

```
remaining = 1024
data = []
while remaining > 0:
    data.append(conn.recv(remaining))
    remaining -= len(data[-1])
data = b''.join(data)
```

] context to keep

Root of Programming Complexity

- We need to keep track of per-connection *contexts*:
  - The number of bytes sent/received
  - Which steps to execute
- We have to deal with not only sockets but also:
  - Synchronization primitives (e.g., locks)
  - Timers & Signals
  - Communication with subprocesses (IPC)
  - Non-std asynchronous I/O events (e.g., CUDA stream callback)
- Current OSes do *not* provide a unified interface for all above.

Our Savior: Coroutines

- The original concept of coroutine
  - Co-operative routines (explicit yields)
  - “Stoppable & resumable” functions (continuation)
- Coroutines + Event loop scheduler
  - Python asyncio
  - C# (.NET Framework 4.5+) async / await
  - C++ boost.coroutine
- Disadvantage
  - Your programming language should support it explicitly.
  - But Python does! ☺

Python asyncio

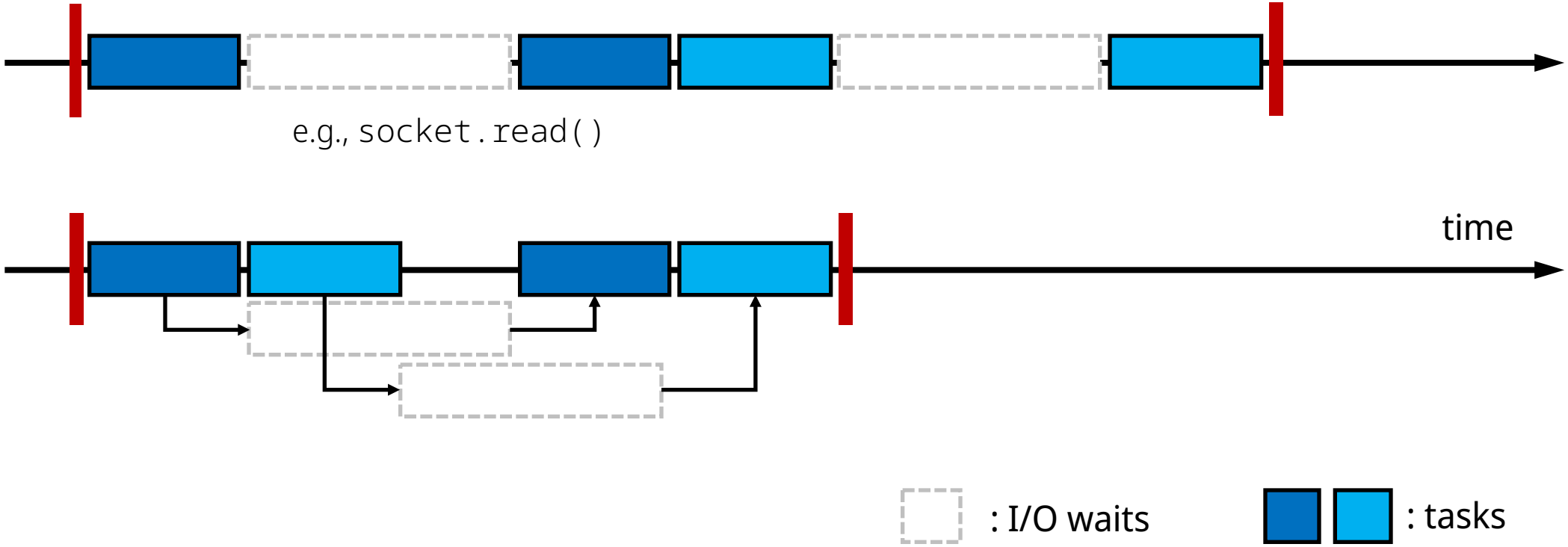
- **PEP-3156** (supplements to **PEP-3153**)
  Asynchronous IO Support Rebooted: the "asyncio" Module
- The Motivation and Goal
  - Existing solutions: asyncore, asynchat, gevent, Twisted, ...
    - Inextensible APIs in existing standard library
    - Lack of compatibility – tightly coupled with what library you use
  - Reusable and persistent event loop API with pluggable underlying implementation
  - Better networking abstraction with Transport and Protocols (like in Twisted)

History of asyncio

- Python 2.2
  - Generators (PEP-255): **`yield`**
- Python 3.3
  - Generator delegation (PEP-380): **`yield from`**
- Python 3.4
  - Event loop integration (PEP-3156): **`asyncio`** package
- Python 3.5
  - Syntactic sugar (PEP-492): **`async`** / **`await`** syntax

asyncio: Why Better?

- Idea: Overlapping blocking I/O reduces total execution time.



asyncio: *Really* Better?

- Important things to get ***actual*** performance gains
  - I/O waits must dominate the total execution time.
  - You should have *many* I/O channels to wait.

May not be true for your specific cases!

- Advantages of asyncio in Python
  - Single-threaded Python apps are *likely to get performance gains* by I/O multiplexing.
  - Even without performance improvements, it is *easier to write programs with concurrent contexts* since they look like sequential codes.
  - It provides *a unified abstraction* for I/O, IPC, timers, and signals.

Generators To asyncio Coroutines

- Generators and generator delegation are the key concepts to understand the asyncio ecosystem.

```
def run():
  for i in myrange(10):
    print(i)
run()
```

```
def srange(n):
  while True:
    time.sleep(0.5)
    if i == n:
      break
    yield i
    i += 1
```

```
class srange:
  def __init__(self, n):
    self.n = n
    self.i = 0

  def __iter__(self):
    return self

  def __next__(self):
    time.sleep(0.5)
    if self.i == self.n:
      raise StopIteration
    i = self.i
    self.i += 1
    return i
```

Generators To asyncio Coroutines

- Generators and generator delegation are the key concepts to understand the asyncio ecosystem.

```
async def run():
  async for i in arange(10):
    print(i)
loop = asyncio.get_event_loop()
loop.run_until_complete(run())
```

```
async def arange(n):
  while True:
    await asyncio.sleep(0.5)
    if i == n:
      break
    yield i
    i += 1
```

not working yet!!

```
class arange:
  def __init__(self, n):
    self.n = n
    self.i = 0

  def __aiter__(self):
    return self

  async def __anext__(self):
    await asyncio.sleep(0.5)
    if self.i == self.n:
      raise StopAsyncIteration
    i = self.i
    self.i += 1
    return i
```

Generators To asyncio Coroutines

- Generators and generator delegation are the key concepts to understand the asyncio ecosystem.
- **`await`** is almost same to **`yield from`** added in Python 3.3.
  - It distinguishes `StopIteration` and `StopAsyncIteration`.
  - They allow transparent two-way communication between the coroutine scheduler (caller of me) and the callee of me.

```
@asyncio.coroutine
def myfunc():
  yield from fetch_data()
```

```
async def myfunc():
  await fetch_data()
```

Generators To asyncio Coroutines

```
async def compose_items(arr):
  while arr:
    data = await fetch_data()
    print(arr.pop() + data)

loop = asyncio.get_event_loop()
loop.run_until_complete(compose_items([1, 2, 3]))
```

The caller may inject exceptions/data into fetch_data().

- `await`-ing in async functions hands over the control to the event loop scheduler.
  - Generator delegation allows the blocking callee to interact with the outer caller transparently to the current context.

Key Things To Learn About asyncio

- Two ways of executing coroutines

```
asyncio.ensure_future(some_coro(...))
loop.create_task(some_coro(...))
```

Non-blocking; returns immediately

```
await some_coro(...)
```

Blocking; returns after finish

  - Always check which functions are coroutines or not.
- **Remember that coroutines are not running in parallel!**
  - They are non-blocking and interleaved manually.
  - Avoid long-running, non-cooperative blocking calls in coroutines.
  - You need to explicitly `cancel_task()` or `loop.stop()` to interrupt a coroutine.

Practical asyncio Tips (1/3)

- Terminating the event loop in different threads
  - Use `loop.call_soon_threadsafe(loop.stop)` where `loop` is the loop of the target thread.
- Debugging unexpected hangs, freezes, etc.
  - Try asyncio's debugging mode (`PYTHONASYNCIODEBUG=1` in env.vars and activate logging for `asyncio`)
  - Use latest Python! (3.5.2 at the time of this talk)
- Use "`async for/with`" whenever available for less code complexity
  - Check out the library manuals (e.g., `aiohttp`)

Practical asyncio Tips (2/3)

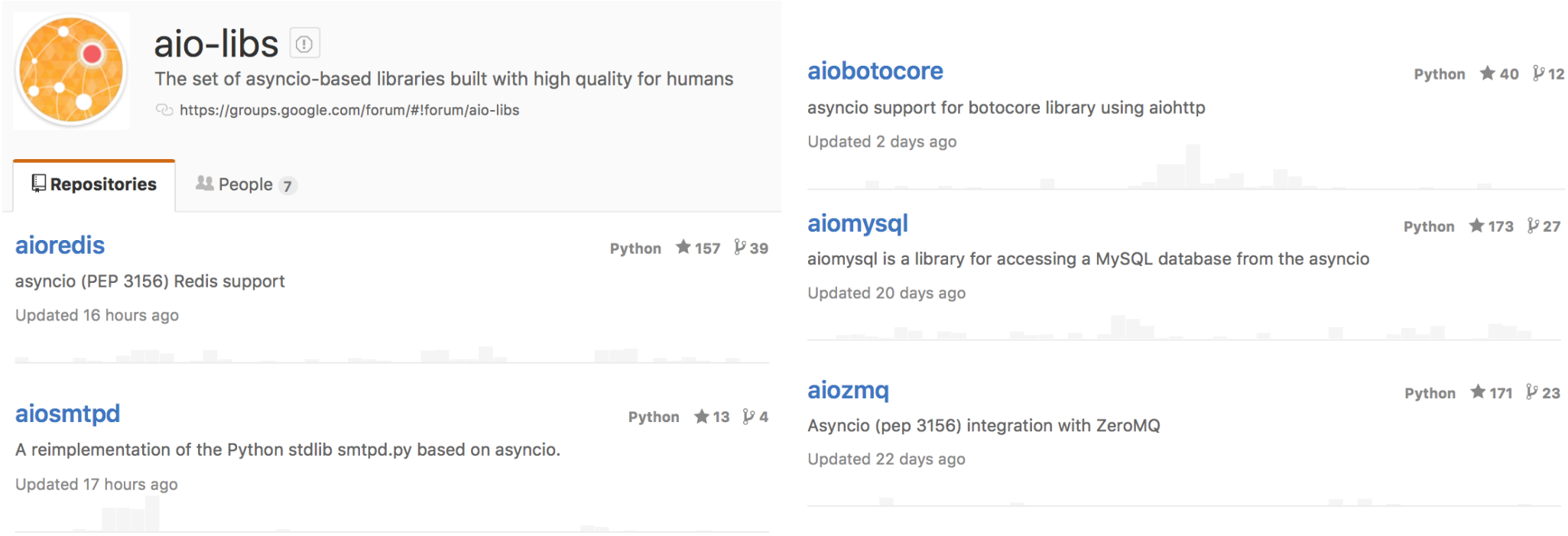
- How to write unit tests for async functions?
  - Simply wrap them with an event loop.

```
class MyTest(unittest.TestCase):
  def setUp(self):
    self.loop = asyncio.new_event_loop()
    asyncio.set_event_loop(self.loop)
  def tearDown(self):
    self.loop.close()
  def test_something(self):
    self.loop.run_until_complete(coro_to_test(...))
    self.assertEqual(...)
```

  - Use a 3rd-party package such as https://github.com/Martiusweb/asynctest.

Practical asyncio Tips (3/3)

- https://github.com/aio-libs
  - First place to look when you need “asyncio-version” of something
  - Reference implementations for those wanting to write asyncio-aware libs



Getting High Performance with asyncio

- Avoid frequent context switching (e.g., polling)
  - It may bring *huge* difference!
- Avoid I/O logic written in Python
  - We all know pure Python loops are slow.
  - It is likely to bring unwanted extra memory copies.

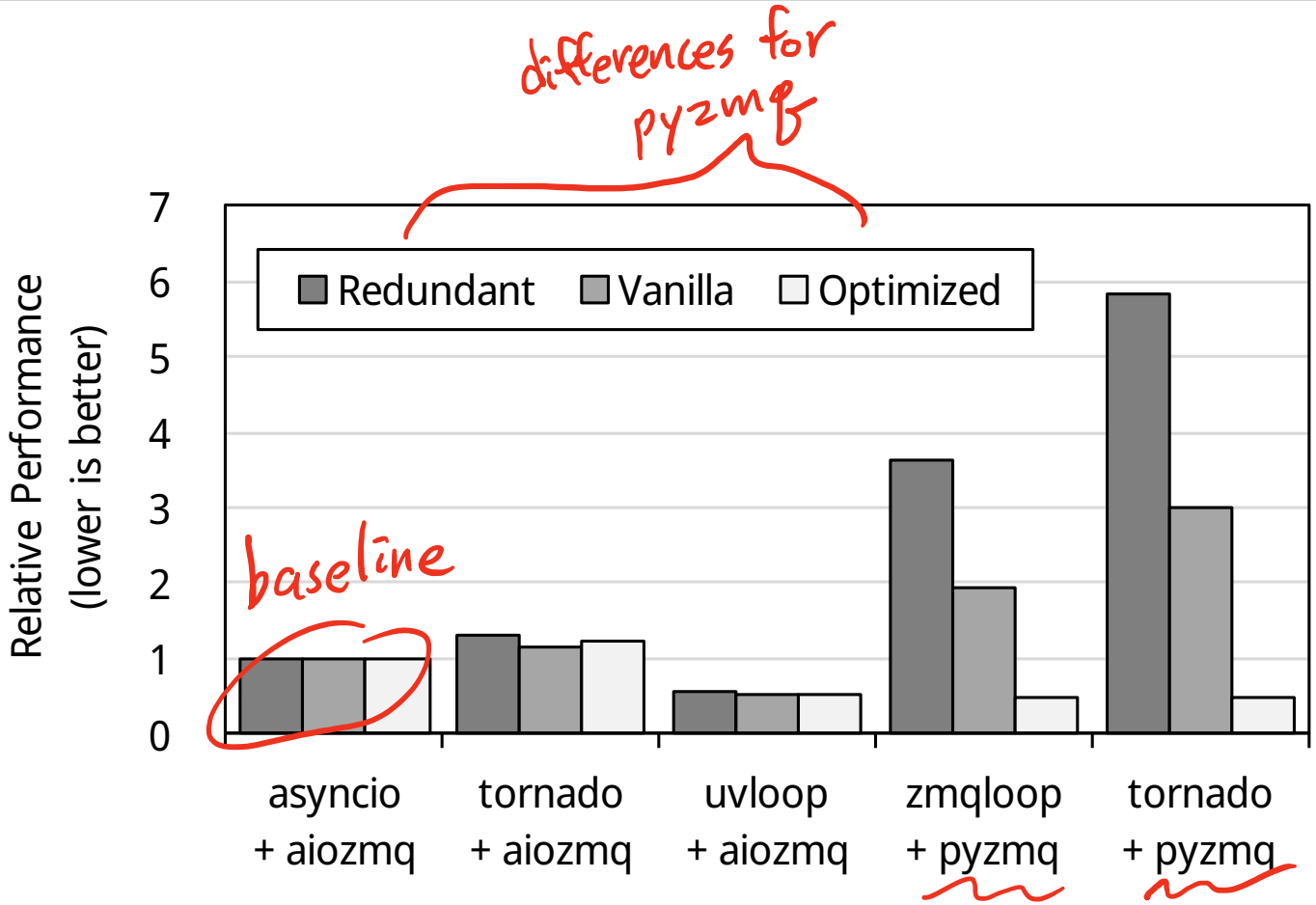
```
remaining = 1024
data = []
while remaining > 0:
    data.append(conn.recv(remaining))
    remaining -= len(data[-1])
data = b''.join(data)
```

- Implement `asyncio.Protocol` instead of using coroutine-based streams.
  - It may add ~5% throughputs.
  - But, don't do this if programming comforts matter (e.g., fast prototyping).
- Use up-to-date, latest libraries (e.g., `uvloop`)
  - The ecosystem is under active development.

How Much Can It Be Different?

- A microbenchmark for ZMQ
  - `aiozmq` *vs.* `pyzmq.asyncio`
  - `asyncio` *vs.* `tornado` *vs.* `zmqloop` *vs.* `uvloop`
  - Workload: two racing push/pull sockets inside a single thread

ZMQ (ZeroMQ): A socket abstraction library that comes with various networking patterns such as queuing and pub/sub using a custom transport extension layer.



https://github.com/achimnol/asyncio-zmq-benchmark

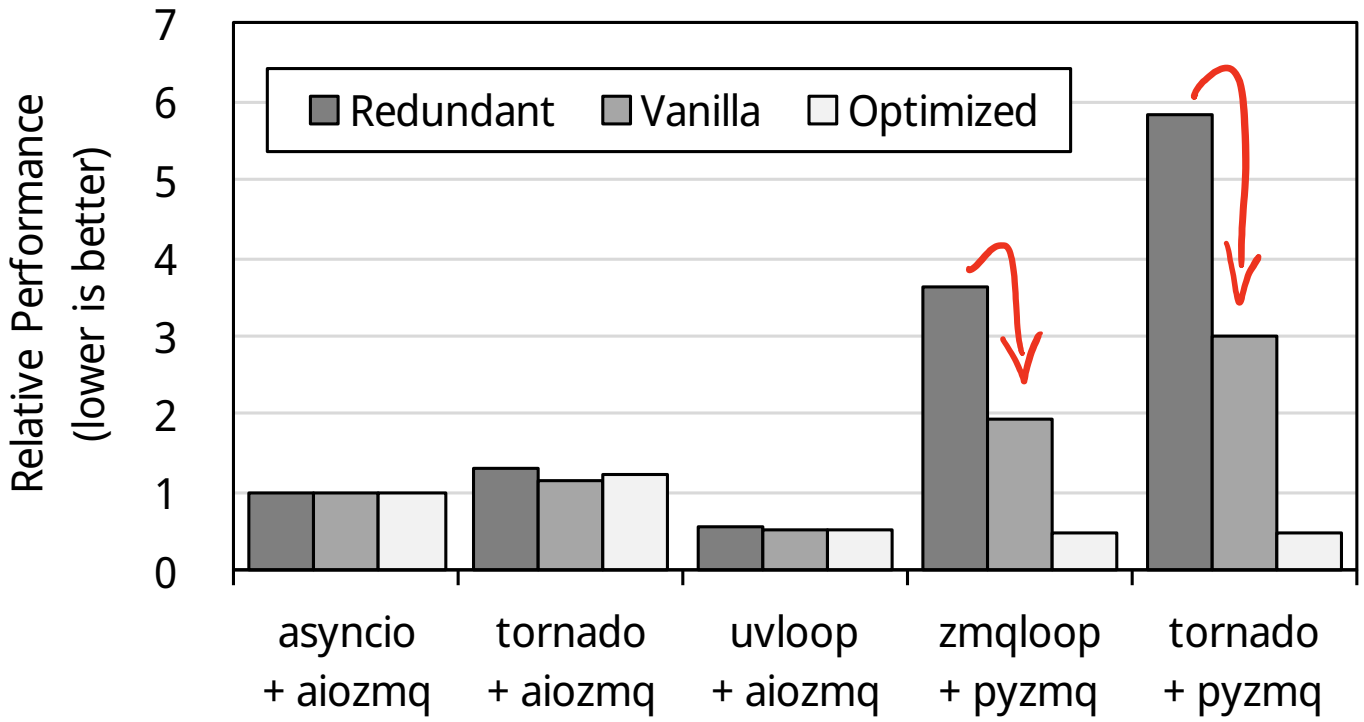
※ pyzmq does not work with uvloop.

How Much Can It Be Different?

- Redundant ➜ Vanilla
  - A mis-implementation with zmqloop & tornado

```
16  16  async def pulling():
17  17      client = ctx.socket(zmq.PULL)
18  18      client.connect('tcp://127.0.0.1:9000')
19      -   poller = zmq.asyncio.Poller()
20      -   poller.register(client, zmq.POLLIN)
21  19      with open(os.devnull, 'w') as null:
22  20          while True:
23      -           events = await poller.poll()
24      -           if client in dict(events):
25      -               greeting = await client.recv_multipart()
26      -               if greeting[0] == b'exit': break
27      -               print(greeting[0], file=null)
        21  +           greeting = await client.recv_multipart()
        22  +           if greeting[0] == b'exit': break
        23  +           print(greeting[0], file=null)
```

Pull Request from Min RK (pyzmq committer)



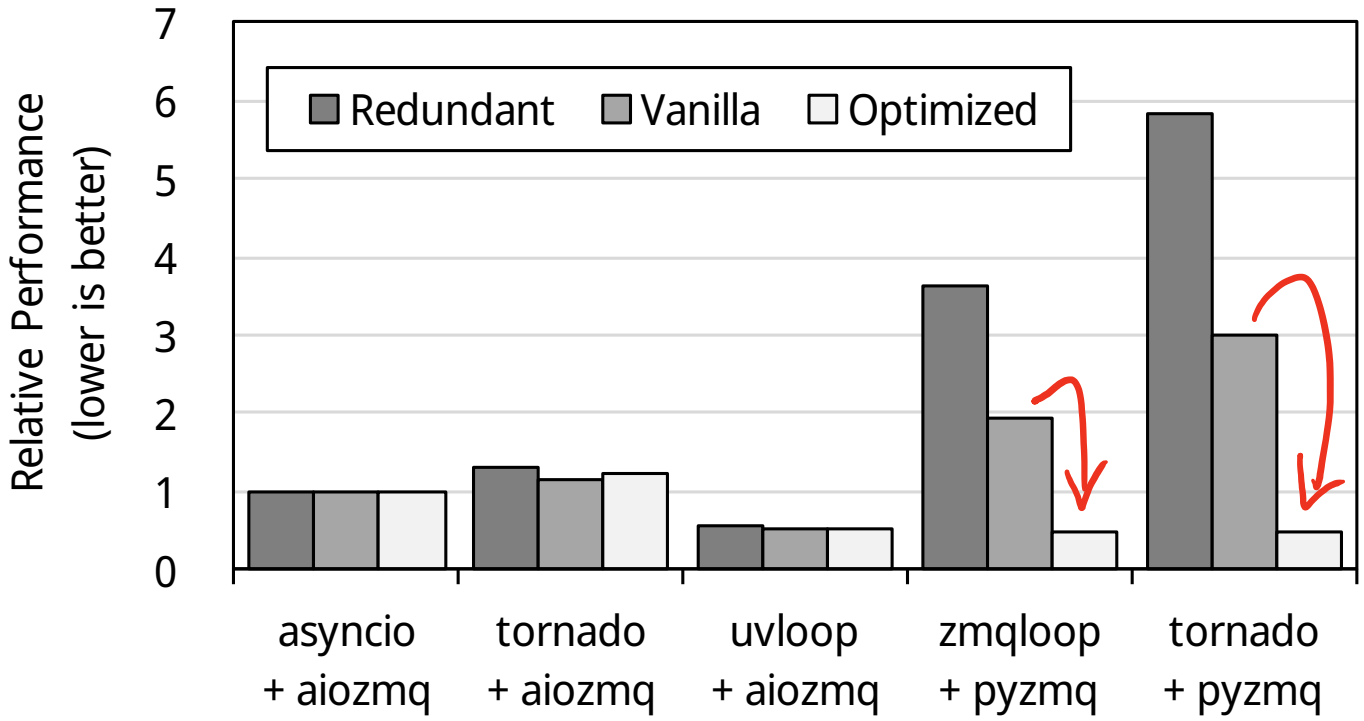
https://github.com/achimnol/asyncio-zmq-benchmark

How Much Can It Be Different?

- Vanilla ➜ Optimized
  - Patching `pyzmq.asyncio` to avoid an extra polling bounce when data is available upon API call.

```
207  208          self._recv_futures.append(
208  209              _FutureEvent(f, kind, kwargs, msg=None)
209  210          )
210       -      self._add_io_state(self._READ)
     211  +      if self.events & POLLIN:
     212  +          # recv immediately, if we can
     213  +          self._handle_recv()
     214  +      if self._recv_futures:
     215  +          self._add_io_state(self._READ)
211  216          return f
```

Excerpt from pyzmq PR#860 by Min RK



https://github.com/achimnol/asyncio-zmq-benchmark

Want More Performance?

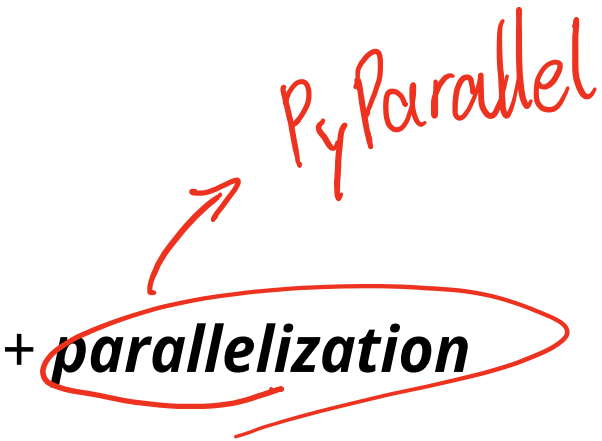
- Use multiple threads or processes.
  (if your app is still I/O-bound!)
  - Try to change threading to **multiprocessing** to avoid GIL.
  - Setting CPU affinity mask may help. (`os.sched_setaffinity`)
  - On *NIX systems: `start_server(..., reuse_port=True)`
- Maybe **PyPy** can boost your app performance.
  (if your app is computation-bound!)
  - *Good news:* Mozilla funds Python 3.5 support in PyPy!
    https://morepypy.blogspot.kr/2016/08/pypy-gets-funding-from-mozilla-for.html
- Most important thing: your **workload** should fit with asyncio.

Want *Even* More Performance? (10+ Gbps)

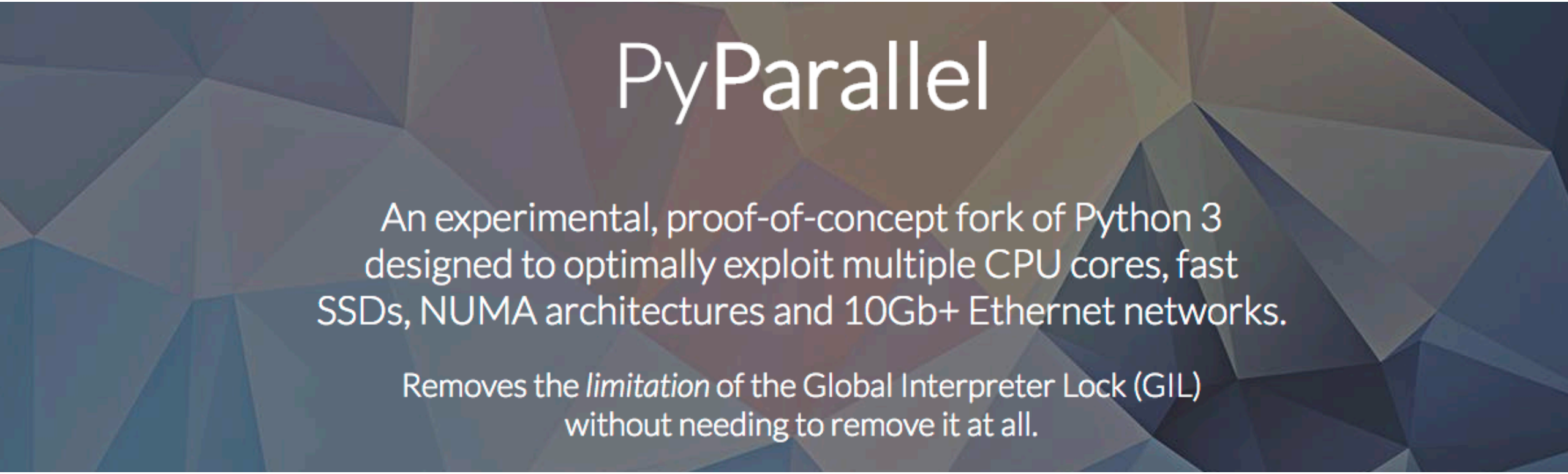
- High-speed networking is intensive!
  - Eight 10 GbE ports ➜ ≥ 88M minimum-sized packets per sec.
  - 2.4 GHz 8-core CPU ➜ ~210 cycles (87 nsec) available per packet
  - cf) x86 `lock`: ~10 nsec, system call: 50 ~ 80 nsec
- Delivering this performance to userspace apps is still challenging!
  - Could a "dynamic" language such as Python keep up?
  - Could the OS network stack (TCP/IP) keep up?
- It is the reality — AWS offers 10 Gbps network interfaces now.

System Programmer's Perspective

- Requirements for high-speed networking (**10+ Gbps**)
  - Zero-copy (DMA buffers directly accessed from userspace)
  - Dedicated DMA packet buffers individually pinned to CPU cores
  - Elimination of generic `malloc()`
    - Usually replaced with custom-optimized memory pools
  - Elimination of synchronization overheads
    - "shared-nothing" architecture
  - NUMA-aware memory allocation
- Core design principles: ***batching*** + ***pipelining*** + ***parallelization***

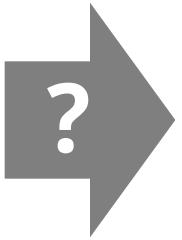


Alternative Approach (for Diversity!)



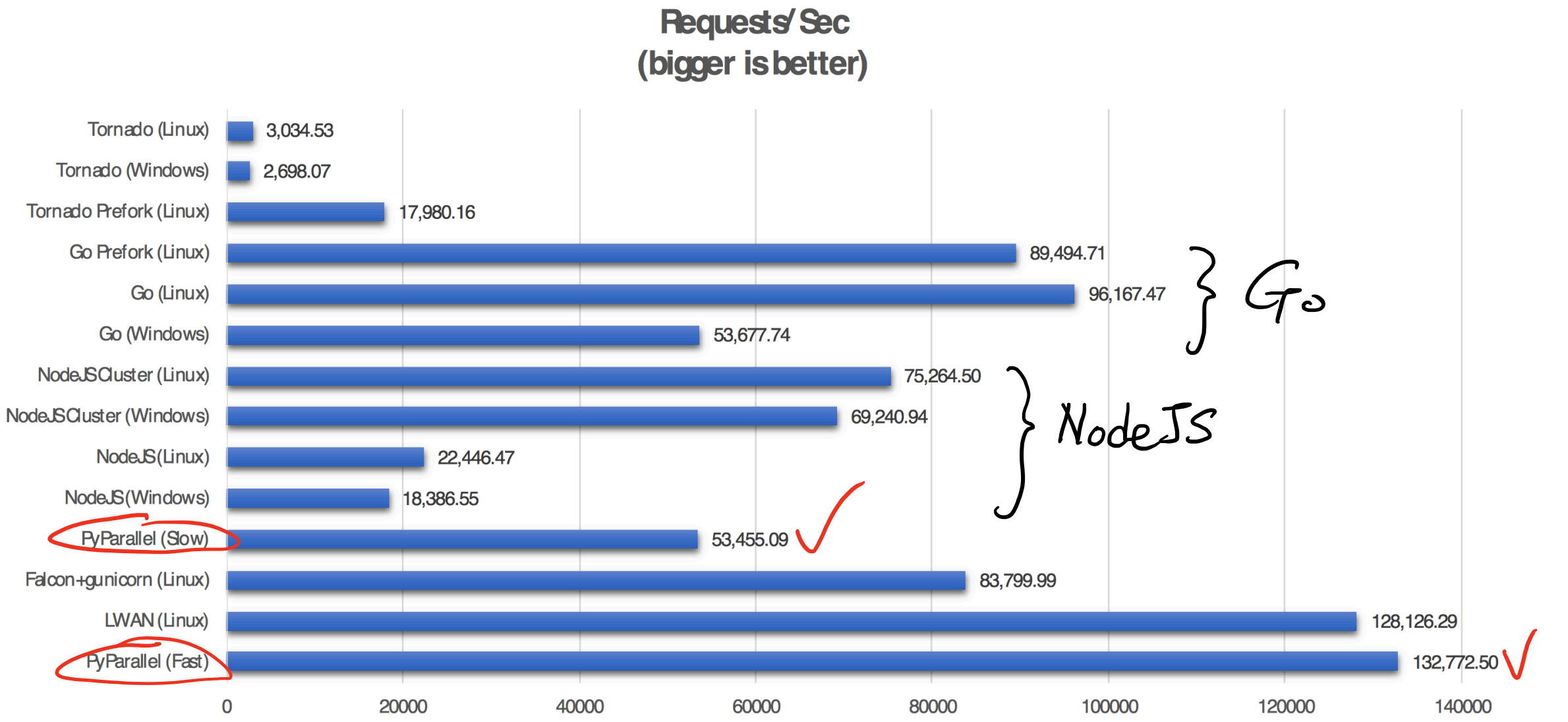
http://pyparallel.org/

Parallelization in Python?



https://www.reddit.com/r/aww/comments/2oagj8/multithreaded_programming_theory_and_practice/

Parallelization in Python!



PyParallel Developer's View

- The current PEP-3156 asyncio (and all other *NIX-based event loops) are *synchronous + non-blocking* I/O instead of actual *asynchronous* I/O!
- The asyncio API is *completion-oriented*. ✓ Good!
- The implementation is *readiness-oriented*. X Bad!
  - Because *NIX systems provides readiness-oriented syscalls for I/O. (select / poll / epoll / kqueue)
- On Windows, we can use completion-oriented, OS-managed APIs called IOCP (IO completion ports).
  - Let's remove obstructions in Python to utilize it.

Completion-oriented vs. Readiness-oriented

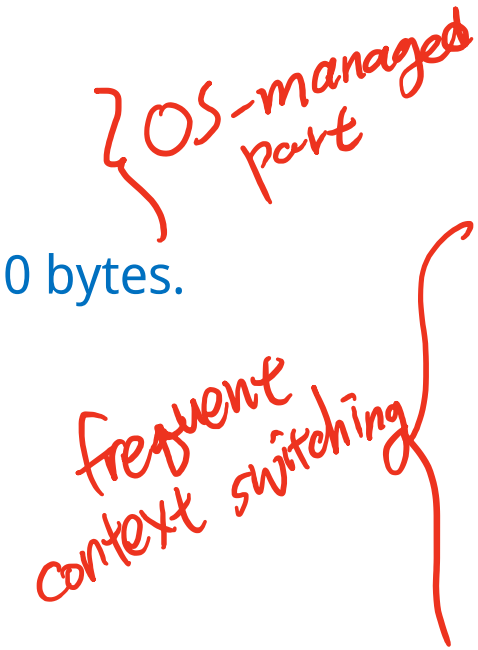
Hey, I have 10 bytes buffer. Please fill it with bytes from this socket.

OK.

...

Here, you got the requested 10 bytes.

Good!



Hey, I have 10 bytes buffer. Please fill it with bytes from this socket.

OK.

Hey, do you have 10 bytes?

I have only 4 bytes. Here they are.

Hey, do you have the remaining 6 bytes?

Not yet. (EAGAIN)

Hey, do you now have those?

Yes, here are another 4 bytes.

Hey, where are the 2 bytes?

...

A modified excerpt from Trent Nelson's talk
https://speakerdeck.com/trent/parallelism-and-concurrency-with-python #47

“True” Parallelization in PyParallel

- (not just like replacing **`threading`** with **`multiprocessing`**…)
- Separation of main thread and parallel context (PCTX)
  - Intercept all thread-sensitive codes. (e.g., `PY_INCREF`)
- GIL & reference counting avoidance
  - If in PCTX, do a thread-safe alternative.
    - Uses a bump memory allocator, with nested heap snapshots to avoid out-of-memory for long-running PCTX programs.
    - All main thread objects are read-only.
    - Main thread and PCTX are mutually exclusively executed.
  - If not in PCTX, do what the original CPython does.

PyParallel Example

- The API resembles asyncio
  - Original name was async but changed due to keyword conflict.

```
import parallel

class Hello:
    def connection_made(self, transport, data):
        return b'Hello, World!\r\n’

    def data_received(self, transport, data):
        return b'You said: ' + data + '\r\n'

server = parallel.server('0.0.0.0', 8080)
parallel.register(transport=server, protocol=Hello)
parallel.run()
```

Summary

- asyncio offers a sweet spot between programmability and high-performance.
  - Advantages come from coroutines enabled by generators
    + clean separation of event loop details and async functions.
  - Pluggable event loops has allowed high-performant 3rd parties such as `uvloop`.
  - For even more performance for multi-cores, we need to *rethink* the underlying OS I/O APIs and Python's GIL with memory mgmt.
  - PyParallel has shown a promising subspace on Windows.
- The Future?

Questions?

OST Room #209 after this talk

Thanks!

Example codes will be available at https://github.com/achimnol soon.

IOCP Model

- Completion-oriented
  - Opposite to *NIX's polling APIs which queries readable/writable states (no matter how much exactly the app wants to read/write).
  - IOCP notifies the app when the given read/write request is done.
- Thread-agnostic I/O
  - Opposite to asyncio (and its relatives) where all I/O requests must be completed by the thread that initiated it.
  - IOCP keeps a set of threads to wake up for completed I/O requests.
    - The number of threads are not limited;
      the number of awoken concurrently threads are limited.
    - Optionally we can use thread affinity for consistent client-thread mapping.