Parallelism Shootout

threads vs. multiple processes vs. asyncio

by Shahriar Tajbakhsh at EuroPython 2015

Me?

Shahriar Tajbakhsh

Software Engineer @ Osper

- github.com/s16h
- twitter.com/STajbakhsh
- linkedin.com/in/STajbakhsh
- shahriar.svbtle.com

What?

We want to download data from **lots and lots*** of URLs stored in a text file and then save that data on our machine.

* We'll actually be using 30 to make demonstration easier and more practical.

How?

Using three different modules:

threading, multiprocessing and asyncio.

Why?

To walk through the mechanics of each approach, then show simple speed benchmarks of the three different approaches.

What's the first rule?

Break down the problem.

Broken Down Problem

1. Read URLs from file
2. Download the content from The Internet™
3. Store the content on our machine

Before we begin…

Reminder

CPU-Bound

A computation where the time for it to complete is determined principally by the speed of the central processor.

I/O-Bound

A computation in which the time it takes to complete it is determined principally by the period spent waiting for input/output operations to be completed.

CPU-Bound or I/O-Bound?

1. Read URLs from file I/O-Bound
2. Download the content I/O-Bound
3. Store the content on our machine I/O-Bound

Just Saying…

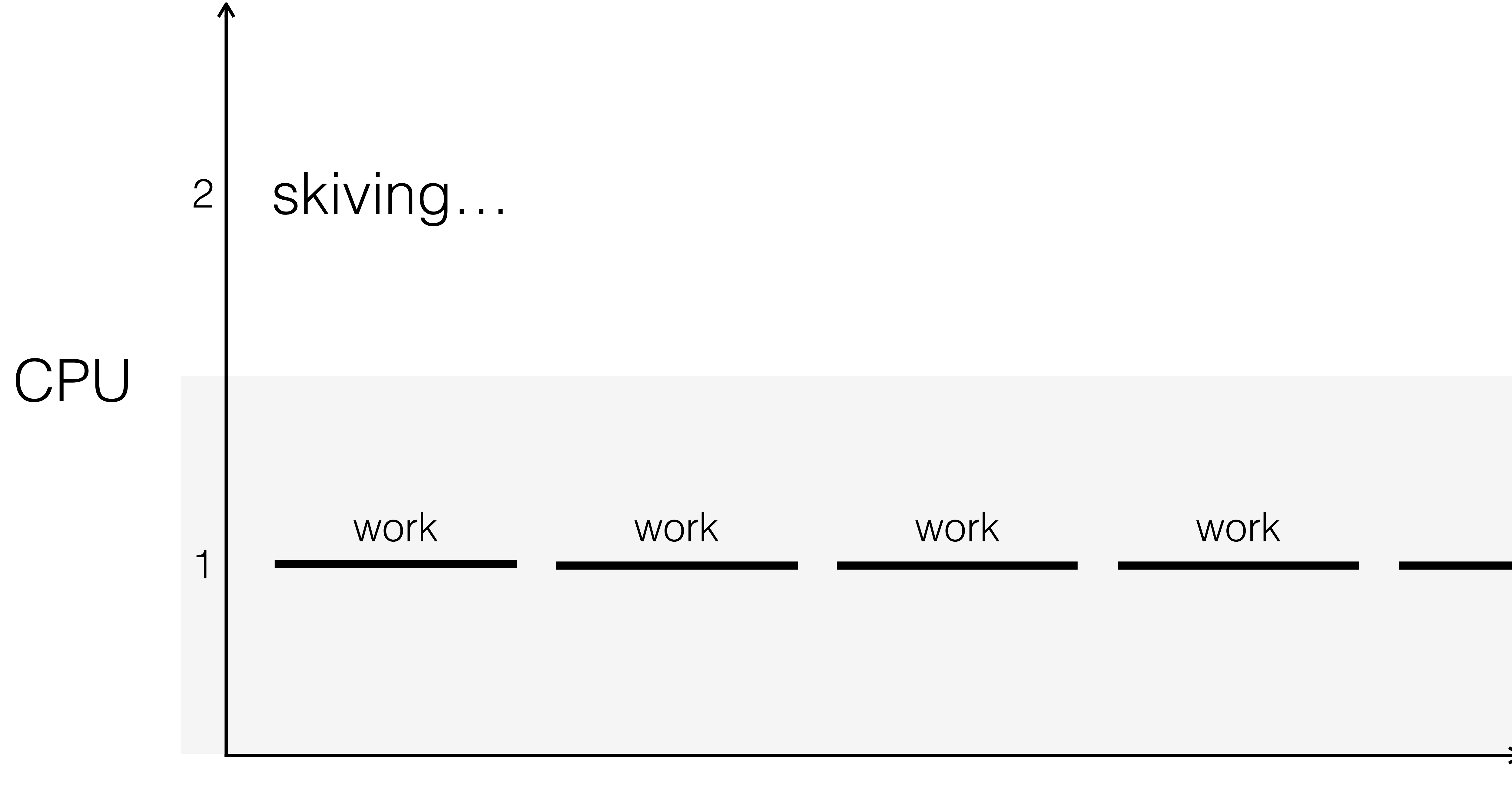
Generally, most* tasks we do are I/O-Bound.

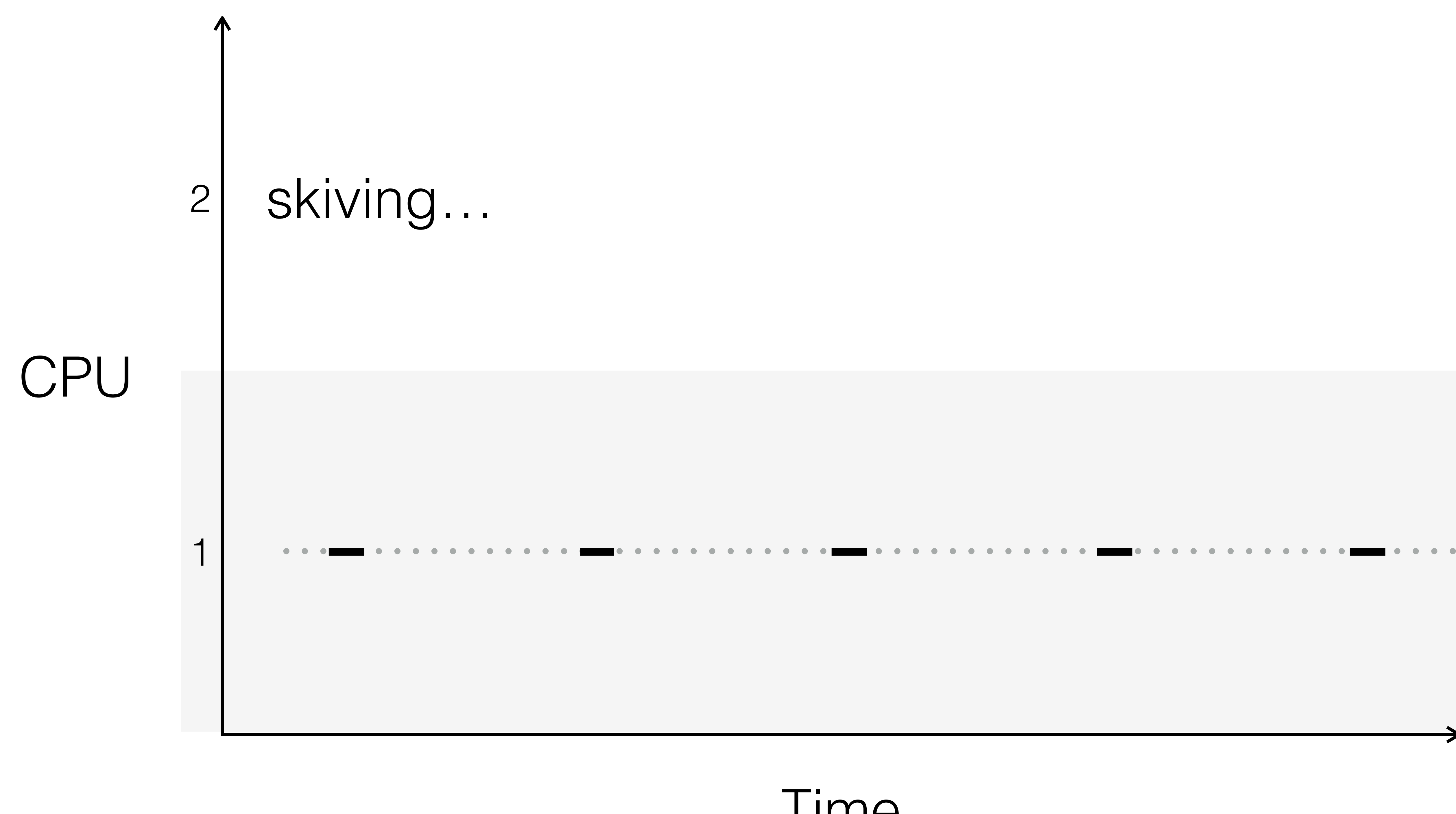
* I haven't statistically looked into this. It's just a guess based on personal experience.

Before we parallelise…

Sequential Approach

```python
import sys

from util import (
    filename_for_url,  # Returns a filename for a URL.
    get_url_content,   # Returns the content at the given URL.
    urls,              # Reads URLs from a file.
    write_to_file      # Writes a string to a file.
)

def main():
    for url in urls('urls.txt'):
        content = get_url_content(url)
        filename = filename_for_url(url, 'downloads')
        write_to_file(filename, content)

if __name__ == '__main__':
    sys.exit(main())
```


CPU
2
skiving…
1
work
work
work
work
Time


2
skiving...
CPU
1
Time
waiting for I/O
CPU working

Benchmark for Sequential Approach

Threading

What Kind of Threads?

Actual real POSIX threads (pthreads) or
Windows threads.

Making Threads

inheriting from `threading.Thread` **or** using `threading.Thread` directly

```
import threading


class MyThread(threading.Thread):
    def run(self):
        print('erm, wow?')


worker = MyThread()
```

```
import threading


def do_work():
    print('erm, wow?')


worker = Thread(target=do_work)
```

Running Threads

call the **run()** method on the **Thread** instance

```python
import threading


class MyThread(threading.Thread):
    def run(self):
        print('erm, wow?')


worker = MyThread()
worker.start()
```

```python
from threading import Thread


def do_work():
    print('erm, wow?')


worker = Thread(target=do_work)
worker.start()
```

Daemons

Threads that run forever need to be made daemonic. Otherwise, when the main thread exits the interpreter will lock.

```
def do_work():
    while True:
        print('Look ma, I never stop!')


worker = threading.Thread(target=do_work, daemon=True)
```

```python
from queue import Queue
from threading import Thread

from sequential_example import do_work
from util import filename_for_url, get_url_content, urls, write_to_file

unvisited_urls = Queue()


def visit_urls():
    while True:
        url = unvisited_urls.get()
        do_work(url)
        unvisited_urls.task_done()


def add_urls_to_queue():
    for url in urls('urls.txt'):
        unvisited_urls.put(url)


def run(number_of_worker_threads):
    add_urls_to_queue()

    for _ in range(number_of_worker_threads):
        worker = Thread(target=visit_urls, daemon=True)
        worker.start()

    unvisited_urls.join()
```

```python
from queue import Queue
from threading import Thread

from sequential_example import do_work
from util import filename_for_url, get_url_content, urls, write_to_file

unvisited_urls = Queue()


def visit_urls():
    while True:
        url = unvisited_urls.get()
        do_work(url)
        unvisited_urls.task_done()


def add_urls_to_queue():
    for url in urls('urls.txt'):
        unvisited_urls.put(url)


def run(number_of_worker_threads):
    add_urls_to_queue()

    for _ in range(number_of_worker_threads):
        worker = Thread(target=visit_urls, daemon=True)
        worker.start()

    unvisited_urls.join()
```

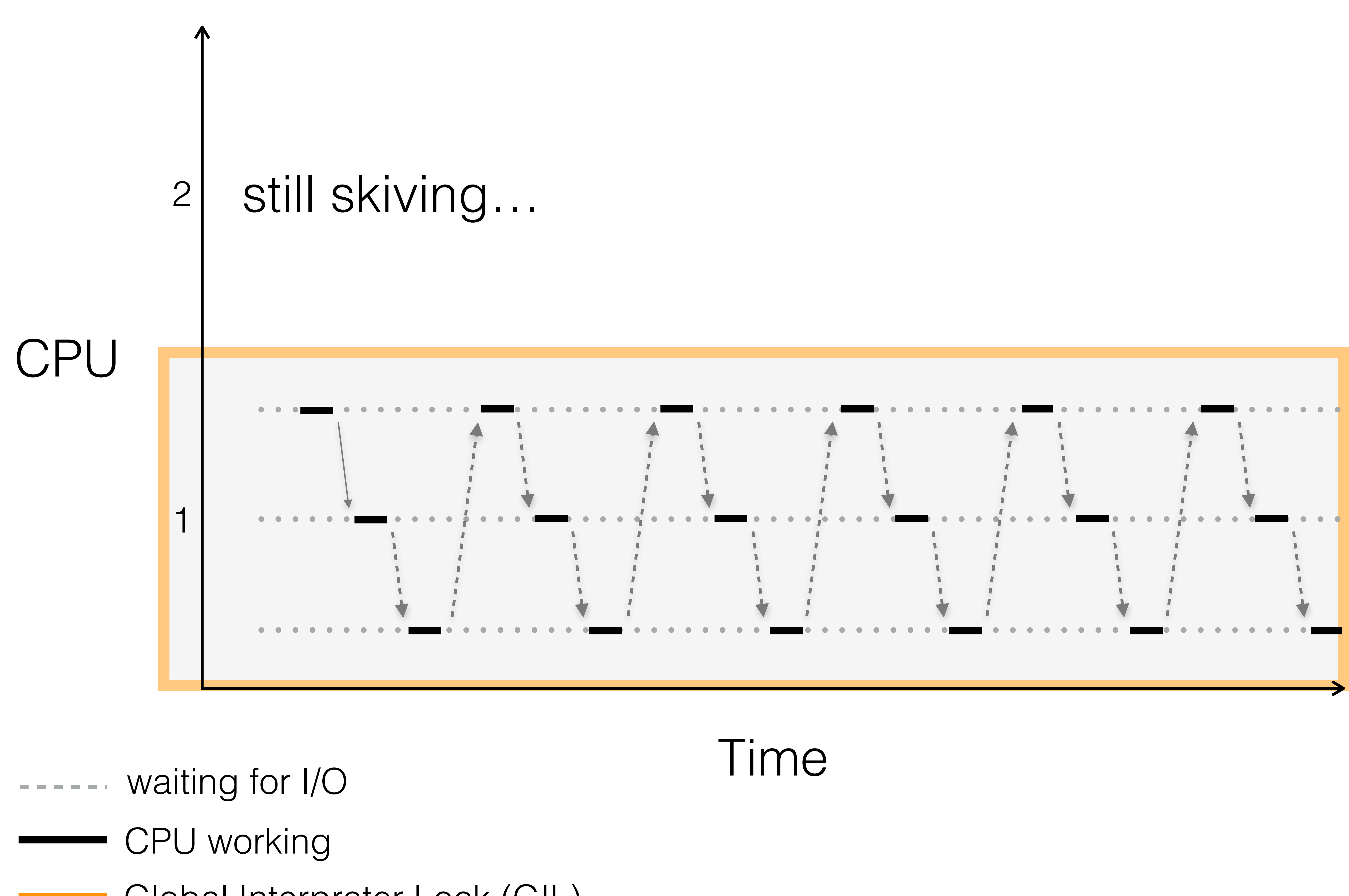
Put all URLs in the queue so different threads can consume them.

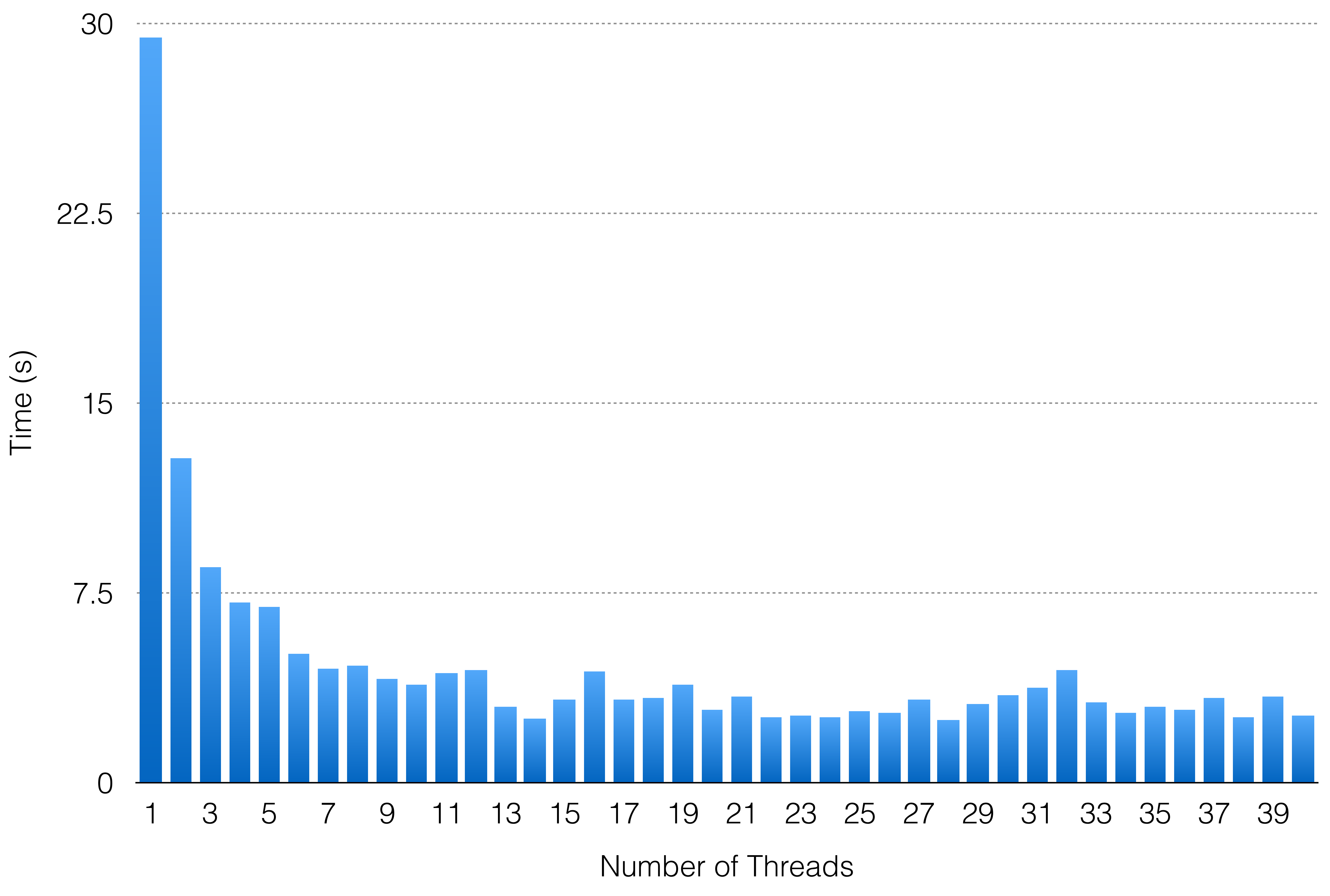
```python
from queue import Queue
from threading import Thread

from sequential_example import do_work
from util import filename_for_url, get_url_content, urls, write_to_file

unvisited_urls = Queue()


def visit_urls():
    while True:
        url = unvisited_urls.get()
        do_work(url)
        unvisited_urls.task_done()


def add_urls_to_queue():
    for url in urls('urls.txt'):
        unvisited_urls.put(url)


def run(number_of_worker_threads):
    add_urls_to_queue()

    for _ in range(number_of_worker_threads):
        worker = Thread(target=visit_urls, daemon=True)
        worker.start()

    unvisited_urls.join()
```

Create daemonic threads.

```python
from queue import Queue
from threading import Thread

from sequential_example import do_work
from util import filename_for_url, get_url_content, urls, write_to_file

unvisited_urls = Queue()


def visit_urls():
    while True:
        url = unvisited_urls.get()
        do_work(url)
        unvisited_urls.task_done()


def add_urls_to_queue():
    for url in urls('urls.txt'):
        unvisited_urls.put(url)


def run(number_of_worker_threads):
    add_urls_to_queue()

    for _ in range(number_of_worker_threads):
        worker = Thread(target=visit_urls, daemon=True)
        worker.start()

    unvisited_urls.join()
```

Do the actual work.


3 Threads
2
still skiving…
CPU
1
Time
waiting for I/O
CPU working
Global Interpreter Lock (GIL)


Benchmark for Threading Approach (with 30 URLs)
Time (s)
30
22.5
15
7.5
0
1
3
5
7
9
11
13
15
17
19
21
23
25
27
29
31
33
35
37
39
Number of Threads

Multiprocessing

multiprocessing

- A package that supports spawning processes using an API similar to the threading module.

- Side-steps the Global Interpreter Lock and allows the programmer to fully leverage multiple processors.

threading example to multiprocessing…

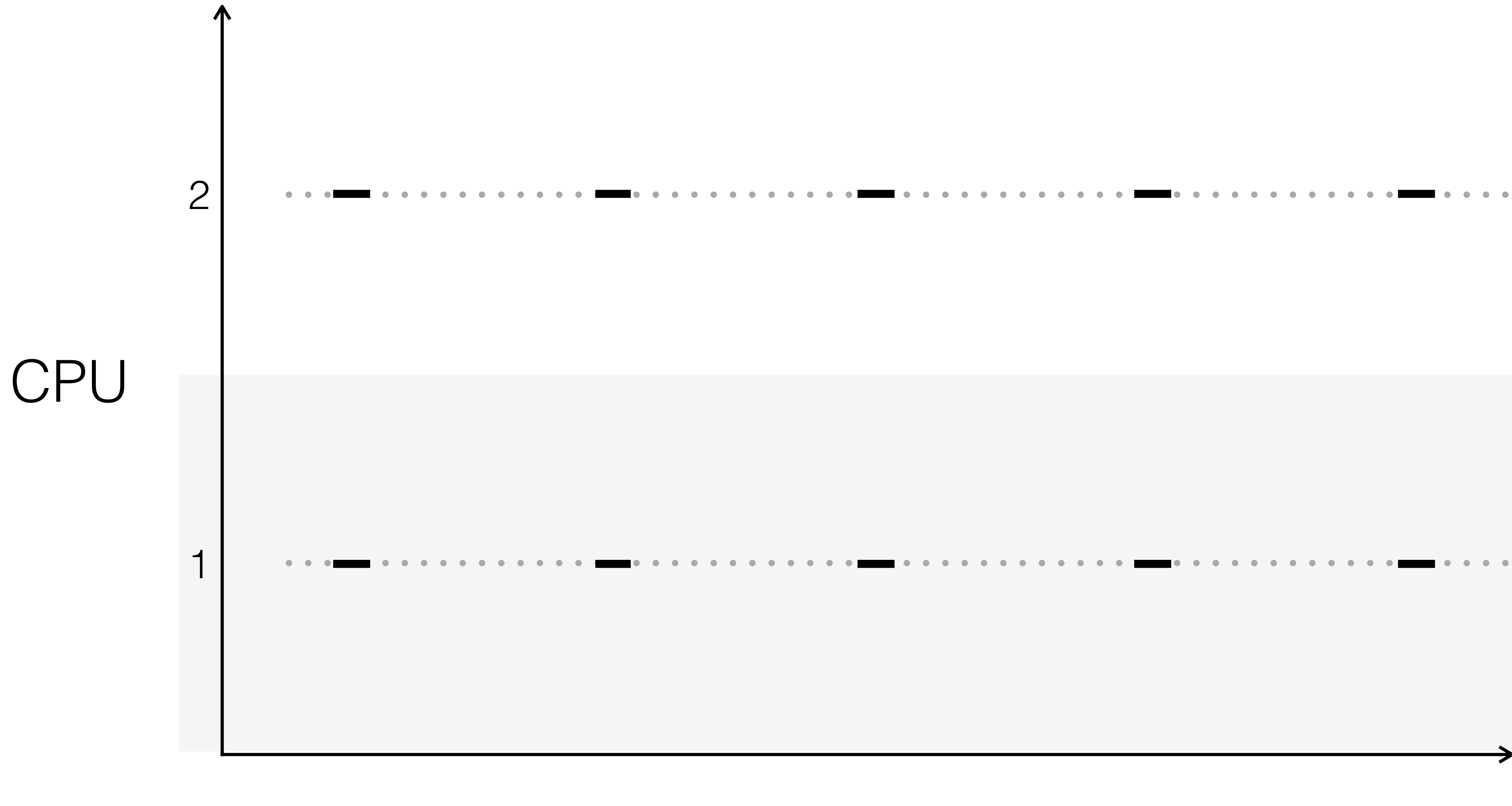
```python
from multiprocessing import Process, Queue

from sequential_example import do_work
from util import filename_for_url, get_url_content, urls, write_to_file

unvisited_urls = Queue()


def visit_urls():
    while True:
        url = unvisited_urls.get()
        do_work(url)
        unvisited_urls.task_done()


def add_urls_to_queue():
    for url in urls('urls.txt'):
        unvisited_urls.put(url)


def run(number_of_worker_threads):
    add_urls_to_queue()

    for _ in range(number_of_worker_threads):
        worker = Process(target=visit_urls, daemon=True)
        worker.start()

    unvisited_urls.join()
```

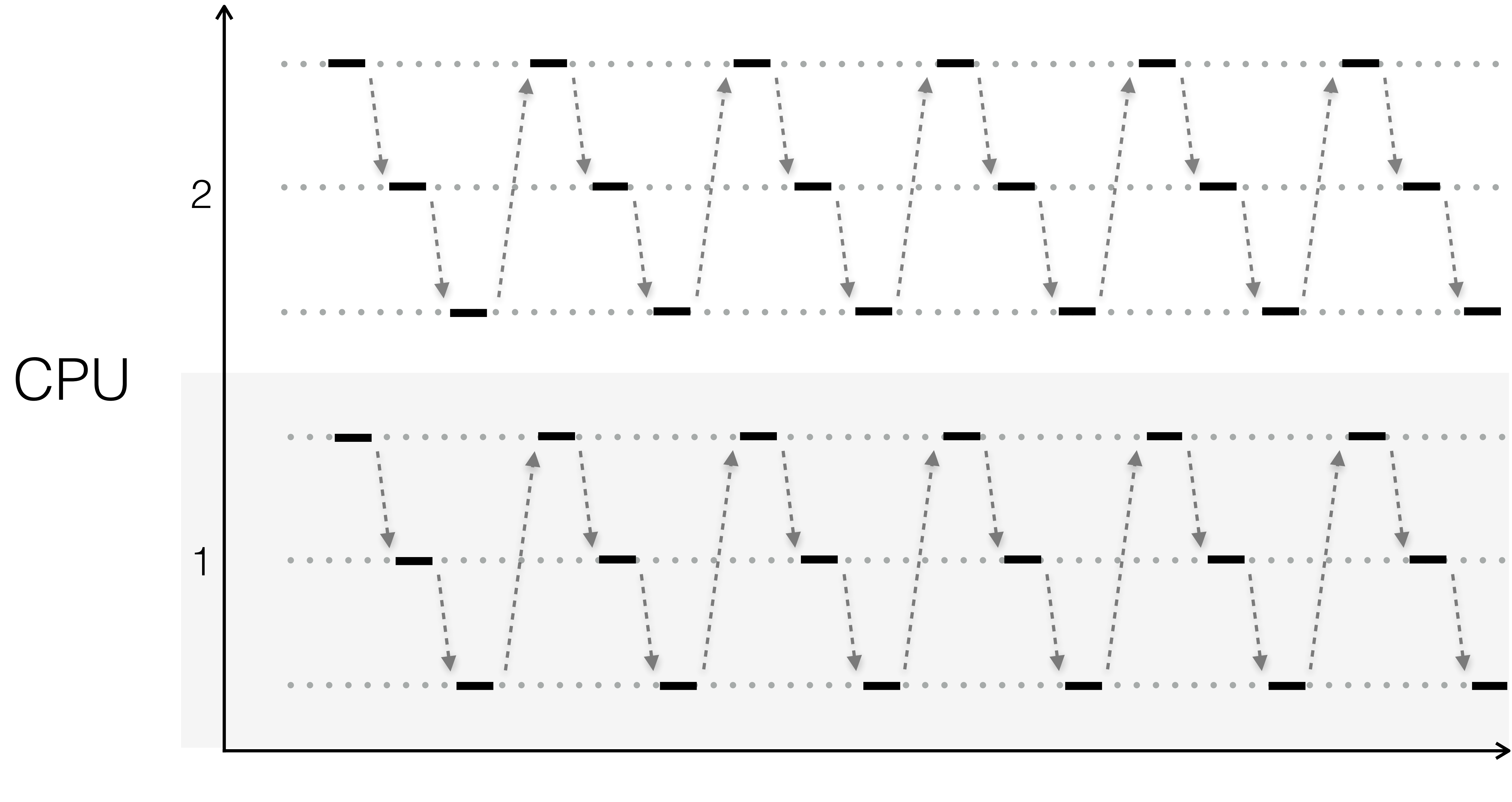
Pool Object

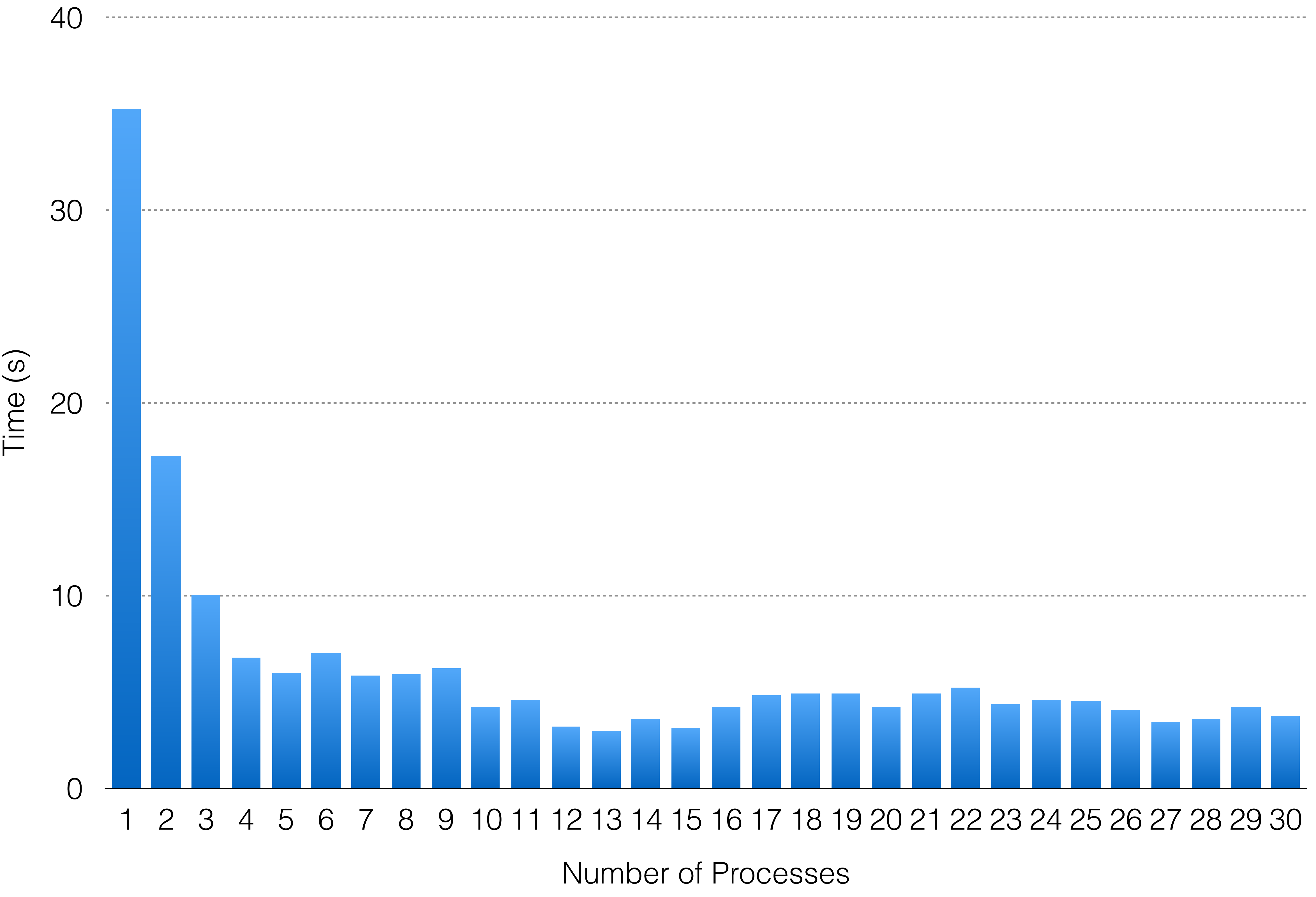
Convenient means of parallelising the execution of a function across multiple input values, distributing the input data across processes (data parallelism).

sequential example to multiprocessing…

```python
from util import filename_for_url, get_url_content, urls, write_to_file

def do_work(url):
    content = get_url_content(url)
    if content:
        filename = filename_for_url(url, 'downloads')
        write_to_file(filename, content)


def run(number_of_worker_processors):
    urls_ = list(urls('urls.txt'))

    with Pool(worker_processes) as pool:
        pool.map(do_work, urls_)
```


2 Processes
CPU
2
1
Time
waiting for I/O
CPU working


6 Processes
2
CPU
1
Time
waiting for I/O
CPU working


Benchmark for Multiprocessing Approach (with 30 URLs)
Time (s)
40
30
20
10
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
Number of Processes

Asyncio

What is asyncio?

- Module added in Python 3.4.
- Provides infrastructure for writing single-threaded concurrent code.
- Low-level; higher level frameworks such as Twisted or Tornado can build on top of it.

Basic asynio Concepts

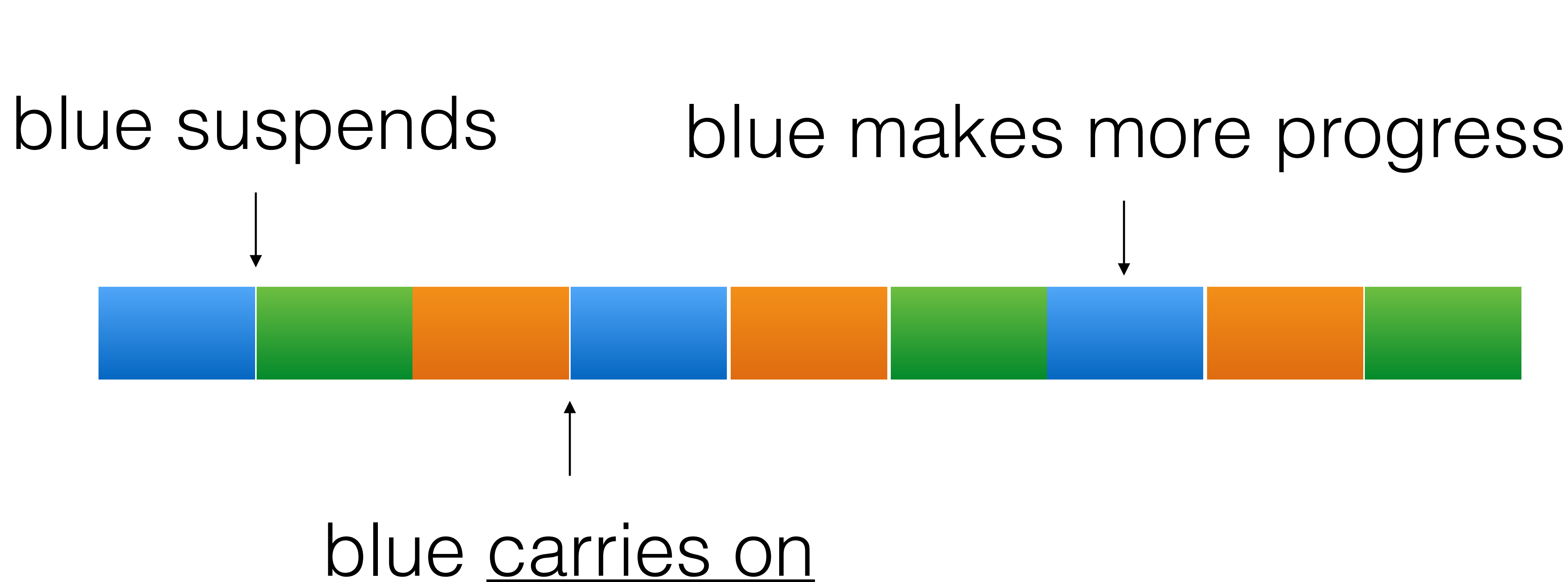
What is a coroutine?

Essentially, a function that can be *suspended* at preset execution points, and *resumed later*, having kept track of its local state.

How is a coroutine used?

- If you have 3 functions to run, on a single thread, you’re forced to run them one-by-one in series.
- In contrast, if you have 3 coroutines, you can interleave their computations.

3 functions run one after the other



Event Loop

The component that is in charge of keeping track of and scheduling all the coroutines that want time on the thread.

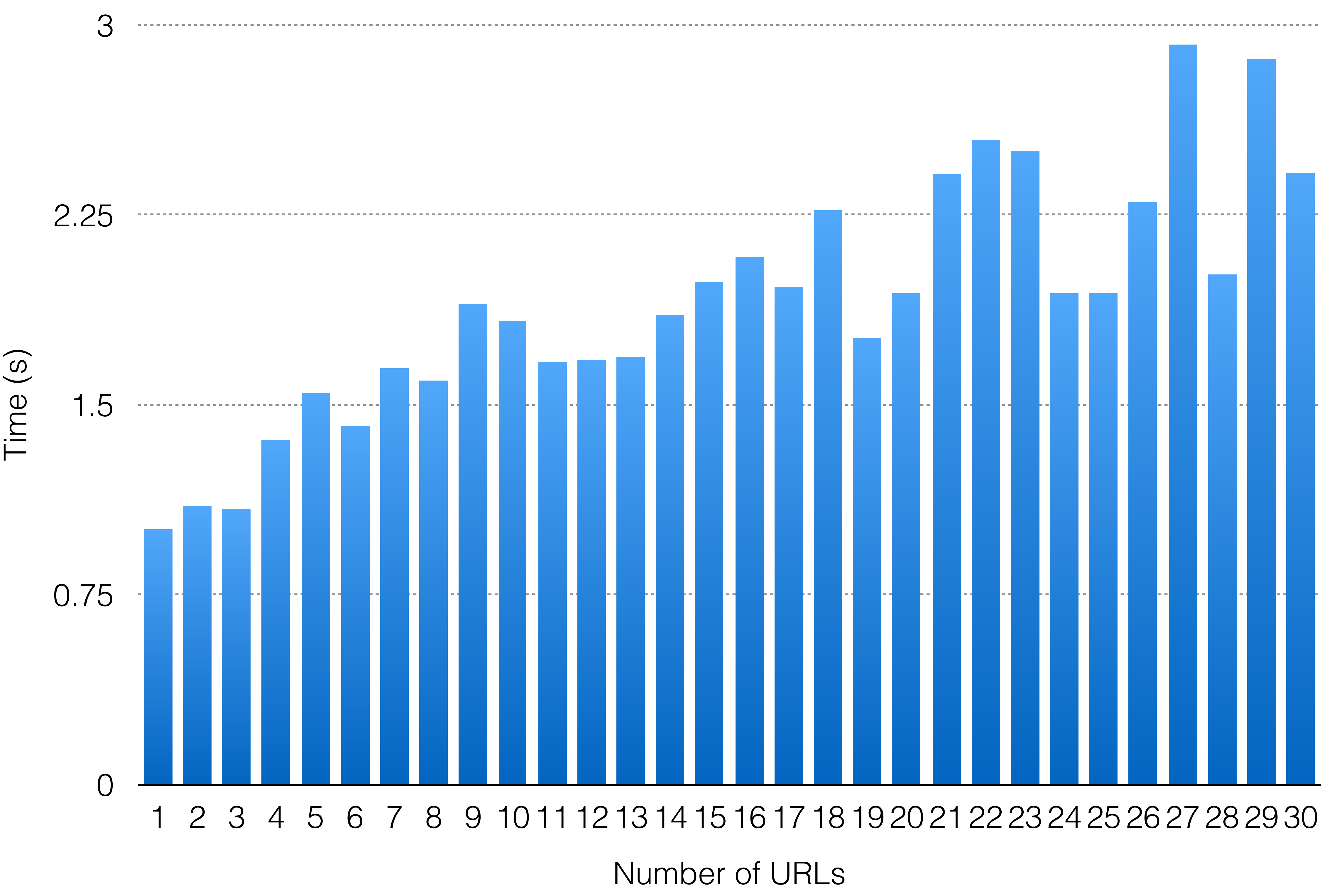
```python
import aiohttp
import asyncio

from util import filename_for_url, urls, write_to_file


@asyncio.coroutine
def get_url_content(url):
    response = yield from aiohttp.request('GET', url)
    return (yield from response.read_and_close())


@asyncio.coroutine
def do_work(url):
    content = yield from asyncio.async(get_url_content(url))
    filename = filename_for_url(url, 'downloads')
    write_to_file(filename, content)


def run():
    coroutines = [do_work(url) for url in urls('urls.txt')]
    event_loop = asyncio.get_event_loop()
    event_loop.run_until_complete(asyncio.wait(coroutines))
    event_loop.close()
```

```python
import aiohttp
import asyncio

from util import filename_for_url, urls, write_to_file


@asyncio.coroutine
def get_url_content(url):
    response = yield from aiohttp.request('GET', url)
    return (yield from response.read_and_close())


@asyncio.coroutine
def do_work(url):
    content = yield from asyncio.async(get_url_content(url))
    filename = filename_for_url(url, 'downloads')
    write_to_file(filename, content)


def run():
    coroutines = [do_work(url) for url in urls('urls.txt')]
    event_loop = asyncio.get_event_loop()
    event_loop.run_until_complete(asyncio.wait(coroutines))
    event_loop.close()
```

```python
import aiohttp
import asyncio

from util import filename_for_url, urls, write_to_file


@asyncio.coroutine
def get_url_content(url):
    response = yield from aiohttp.request('GET', url)
    return (yield from response.read_and_close())


@asyncio.coroutine
def do_work(url):
    content = yield from asyncio.async(get_url_content(url))
    filename = filename_for_url(url, 'downloads')
    write_to_file(filename, content)


def run():
    coroutines = [do_work(url) for url in urls('urls.txt')]
    event_loop = asyncio.get_event_loop()
    event_loop.run_until_complete(asyncio.wait(coroutines))
    event_loop.close()
```

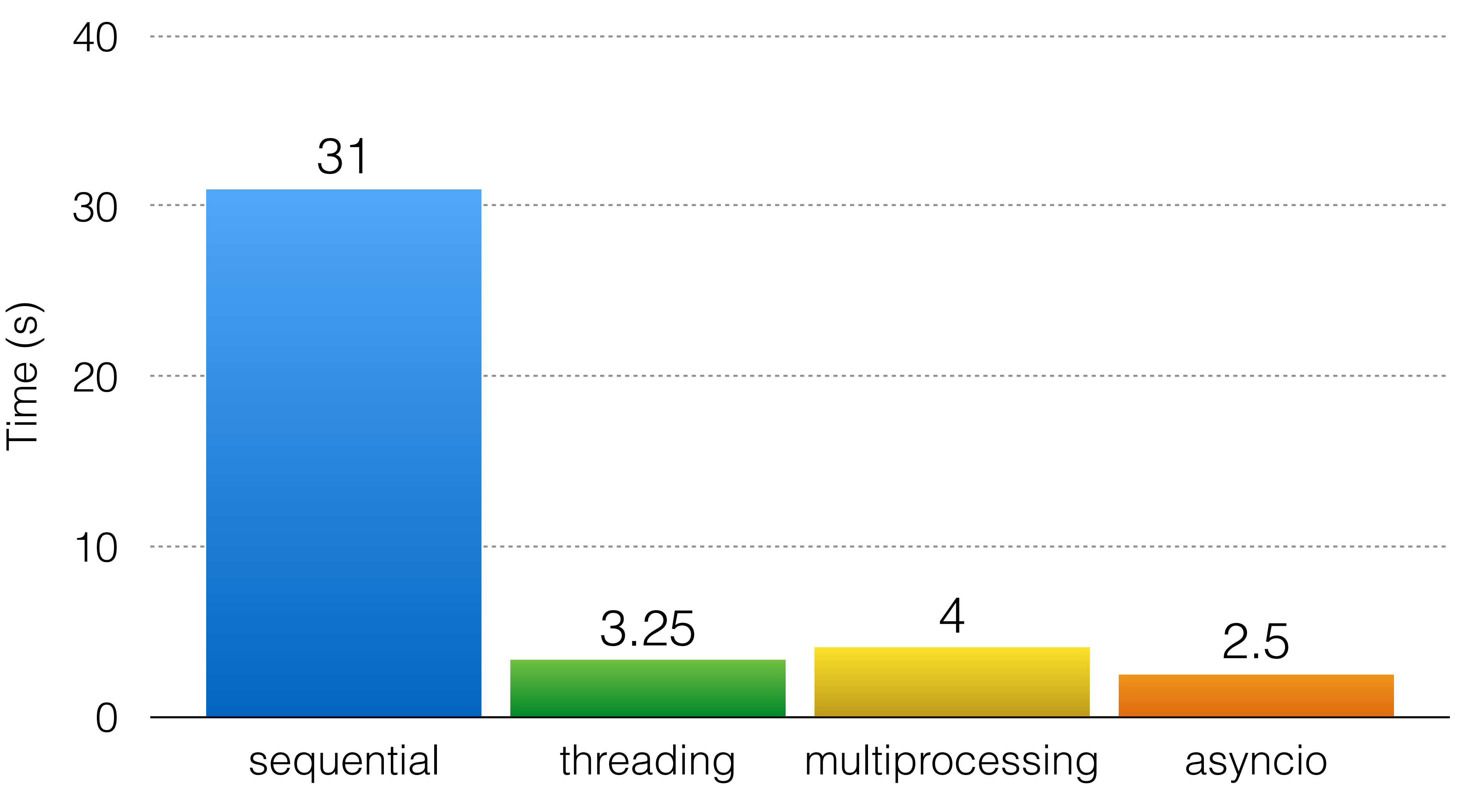
```python
import aiohttp
import asyncio

from util import filename_for_url, urls, write_to_file


@asyncio.coroutine
def get_url_content(url):
    response = yield from aiohttp.request('GET', url)
    return (yield from response.read_and_close())


@asyncio.coroutine
def do_work(url):
    content = yield from asyncio.async(get_url_content(url))
    filename = filename_for_url(url, 'downloads')
    write_to_file(filename, content)


def run():
    coroutines = [do_work(url) for url in urls('urls.txt')]
    event_loop = asyncio.get_event_loop()
    event_loop.run_until_complete(asyncio.wait(coroutines))
    event_loop.close()
```

```python
import aiohttp
import asyncio

from util import filename_for_url, urls, write_to_file


@asyncio.coroutine
def get_url_content(url):
    response = yield from aiohttp.request('GET', url)
    return (yield from response.read_and_close())


@asyncio.coroutine
def do_work(url):
    content = yield from asyncio.async(get_url_content(url))
    filename = filename_for_url(url, 'downloads')
    write_to_file(filename, content)


def run():
    coroutines = [do_work(url) for url in urls('urls.txt')]
    event_loop = asyncio.get_event_loop()
    event_loop.run_until_complete(asyncio.wait(coroutines))
    event_loop.close()
```


Benchmark for asyncio Approach
Time (s)
3
2.25
1.5
0.75
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
Number of URLs

Drum roll

Speed Comparison



Conclusion?

I prefer not to conclude when it comes to parallelism.

Who was I?

Shahriar Tajbakhsh
Software Engineer @ Osper

github.com/s16h
twitter.com/STajbakhsh
linkedin.com/in/STajbakhsh
shahriar.svbtle.com

Code and Other Resources

Will be at

https://github.com/s16h/EuroPython-2015

after the talk.

Q&PA

Questions and Possible Answers!