Choosing **Concurrency** and **Parallelism** for Your **Python Projects**

Muhammad Shalahuddin Yahya Sunarko
AI Technical Lead - Qlue Performa Indonesia

“Dealing with concurrency becomes hard when we lack the ‘working knowledge’ and best practices are not followed.” – **Ramith Jayasinghe, Experienced Software Engineer.**

Muhammad Shalahuddin Yahya Sunarko

AI Technical Lead

- Leading AI and ML development at Qlue
- Interested in AI and High Performance Computing
- NVIDIA Deep Learning Institute certified

qlue

Qlue is a **comprehensive smart city ecosystem company** which is based on AI (Artificial Intelligence) and IoT (Internet of Things) technology innovation. Various Qlue solutions help decision makers in making more effective and efficient business decision.

We also create a platform that **connect government and its citizens**, and visualize the data gathered from our various technology implementations.

Vision

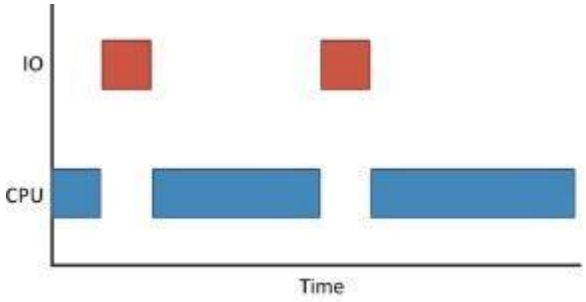
Accelerating positive changes worldwide

Table of Contents

The Fundamental **Concepts**

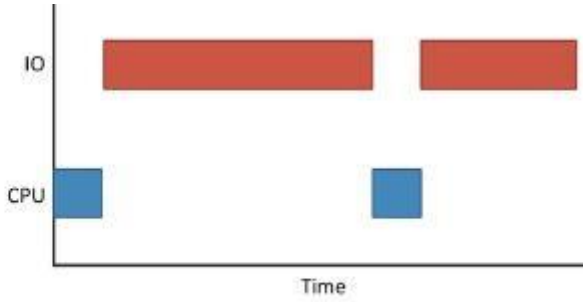
CPU-bound vs I/O-bound

- Waktu utilisasi CPU > waktu menunggu I/O
- Contoh: AI/ML computation program



(a) CPU-bound application

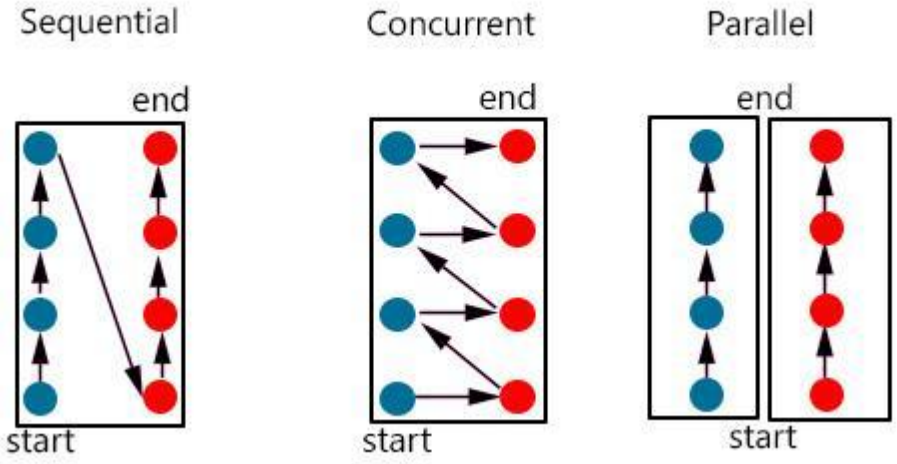
- Waktu utilisasi CPU < waktu menunggu I/O
- Contoh: web application program



(b) IO-bound application

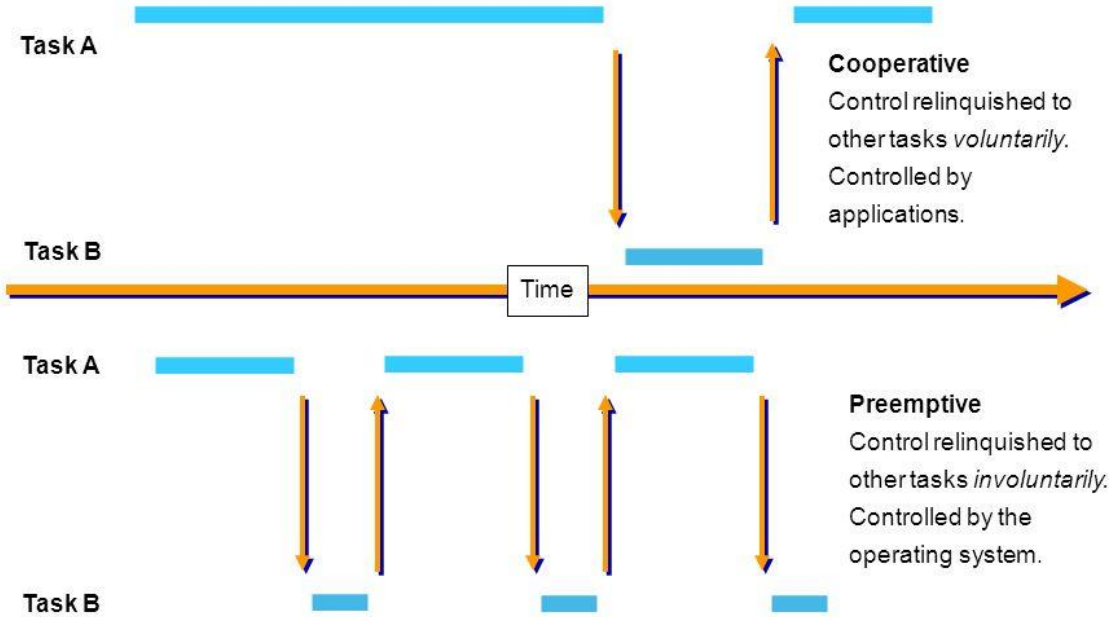
Sumber gambar: https://ars.els-cdn.com/content/image/1-s2.0-S0045790616302270-gr1.jpg

Concurrency and Parallelism Demystified



Sumber gambar: http://www.dietergalea.com/parallelism-concurrency/ ; dimodifikasi

Multitasking: Cooperative vs Pre-emptive



Sumber gambar: https://slideplayer.com/slide/8851057/

Concurrency in **(C)Python**

IronPython

“Python is actually a specification for a language that can be implemented in many different ways.” – Kenneth Reitz & Real Python

Race Condition

Thread 1	Thread 2		Nilai integer
			0
baca nilai		←	0
	baca nilai	←	0
tambahkan 1			0
	tambahkan 1		0
tulis kembali		→	1
	tulis kembali	→	1

Race Condition terjadi ketika terdapat 2 *thread* yang melakukan baca dan tulis pada suatu variabel secara bersamaan, sehingga menyebabkan *bug* pada saat eksekusi.

Thread 1	Thread 2		Nilai integer	Lock state
			0	unlocked
acquire lock			0	unlocked
lock acquired			0	locked
	acquire lock		0	locked
baca nilai		←	0	locked
tambahkan 1			0	locked
tulis kembali		→	1	locked
release lock			1	locked
lock released			1	unlocked
	lock acquired		1	locked
	baca nilai	←	1	locked
	tambahkan 1		1	locked
	tulis kembali	→	2	locked
	release lock		2	locked
	lock released		2	unlocked

Masalah ini dapat diatasi dengan salah satu cara sinkronisasi antar *thread*, yaitu menggunakan *lock*.

Global Interpreter Lock in (C)Python

- Manajemen memori (C)Python: *reference counting*
- Race Condition pada mekanisme reference counting: *memory leak* atau memori dibebaskan sedangkan sebenarnya masih digunakan.
- Solusi (C)Python: sebuah Global Interpreter Lock (GIL) untuk seluruh *thread* pada suatu *process* dalam interpreter saat eksekusi baris-baris kode perintah.

Threading in **(C)Python**

Threading in (C)Python

- *Built-in package* untuk “multithreading” *concurrency*
- Hanya menggunakan 1 *core* CPU, meskipun prosesor *multi-core*
- Kurang cocok untuk CPU-*bound*, lebih cocok untuk I/O-*bound*

Class inheritance

```
import threading

class ThreadKu(threading.Thread):
  def __init__(self, *args, **kwargs):
    threading.Thread.__init__(self)
    ...

  def run(self):
    ...

if __name__ == "__main__":
  ...
  t_0 = ThreadKu(*args, **kwargs)
  t_0.start()
  ...
  t_0.join()
  ...
```

Passing callable

```
import threading

def function_ku(*args, **kwargs):
  ...

if __name__ == "__main__":
  ...
  t_0 = threading.Thread(
    target=function_ku,
    args=args,
    kwargs=kwargs)
  t_0.start()
  ...
  t_0.join()
  ...
```

Example CPU-bound Program

```python
import time
import numpy as np

def count_ops(job_duration: float, array_s: int = 64) -> int:
  ops_count = 0
  start_time = time.perf_counter()
  while time.perf_counter() - start_time < job_duration:
    a = np.random.rand(array_s, array_s)
    x = np.random.rand(array_s, array_s)
    b = np.random.rand(array_s, array_s)
    _ = np.matmul(a, x) + b
    ops_count += 1
  return ops_count
```

CPU-bound: Sequential vs Threading

In [2]:
```
# Versi sekuensial untuk perbandingan
!python3 src/python/hello_world.py
```

```
Finished with 9.854k operations, about 9.854k operation per second
```

In [4]:
```
# Berikut dapat dilihat contoh untuk cara threading yang pertama:
!python3 src/python/hello_world_mt.py
```

```
Finished all jobs, totalling 8.922k operations, about 8.912k operations per second
```

In [6]:
```
# Berikut dapat dilihat contoh untuk cara threading yang kedua:
!python3 src/python/hello_world_mt_2.py
```

```
Finished all jobs, totalling 8.613k operations, about 8.604k operations per second
```

Example I/O-bound Program

```
import requests

def fetch_url(session: requests.Session, url: str):
  with session.request("GET", url) as response:
    data = response.content
    return data

def main():
  urls = ["http://localhost:5000", ...]
  with requests.Session() as sess:
    urls_data = []
    for url in urls:
      data = fetch_url(session=sess, url=url)
      urls_data.append(data)
```

I/O-bound: Sequential vs Threading

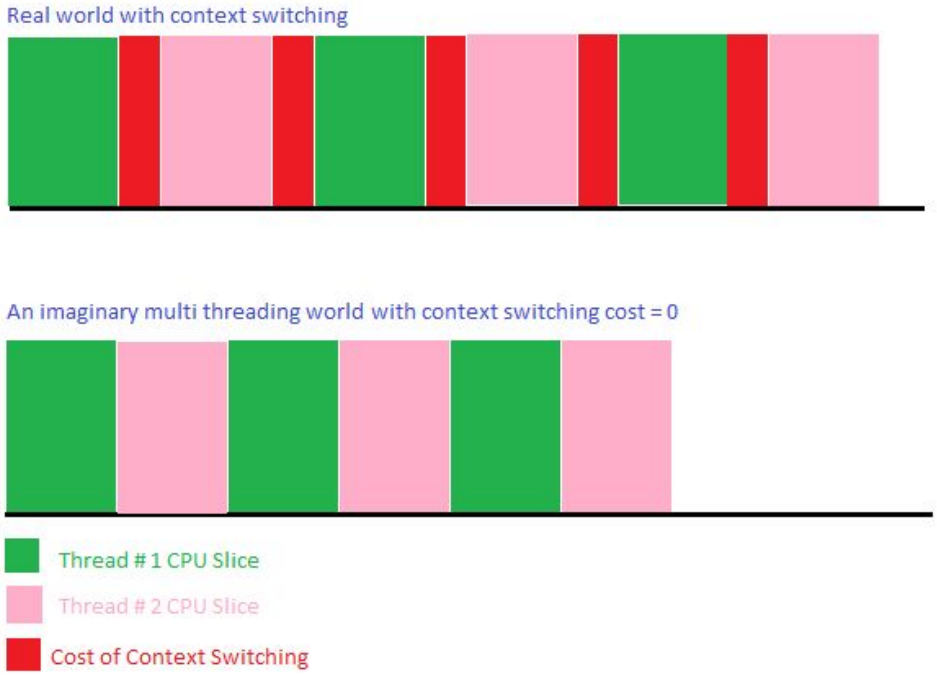
```
In [9]: # Versi sekuensial untuk perbandingan
        !python3 src/python/hello_world_io.py

        Finished all jobs in 588.41 ms

In [11]: # Versi multithreaded
         !python3 src/python/hello_world_io_mt.py

         Finished all jobs in 856.66 ms
```

Context Switching



Sumber gambar: https://www.codeproject.com/Articles/1083787/Tasks-and-Task-Parallel-Library-TPL-Multi-threadin

Asyncio in **(C)Python**

Asyncio in (C)Python

- *Built-in package* untuk *asynchronous* (*concurrent*) I/O
- Hanya menggunakan 1 *thread* dan 1 *core* CPU
- Beberapa konsep penting dalam asyncio: *event loop*, *coroutine*, dan *future*.

Asyncio in (C)Python

- *Event loop*: perangkat eksekusi utama yang disediakan oleh asyncio, tempat di mana seluruh *task* berjalan.
- *Coroutine*: function dengan keyword async def yang eksekusi kodenya dapat berhenti secara kooperatif sebelum return. Keyword await digunakan dalam *coroutine* untuk *yield* (menyerahkan) kontrol eksekusi CPU saat menunggu hasil.
- *Future*: saat *coroutine function* dipanggil, belum terjadi eksekusi apapun dan didapatkan objek *future*. Diperlukan await untuk mendapatkan hasil dari *future*.

Example I/O-bound Program (Asyncio)

```
import asyncio
import aiohttp

async def fetch(session: aiohttp.ClientSession, url: str):
  async with session.request("GET", url) as response:
    data = await response.read()
    return data

async def async_main():
  urls = ["http://localhost:5000", ...]
  async with aiohttp.ClientSession() as sess:
    futs = [fetch(session=sess, url=url) for url in urls]
    urls_data = await asyncio.gather(*futs)
    for data, url in zip(urls_data, urls):
      urls_data.append(data)

def main():
  loop = asyncio.get_event_loop()
  loop.run_until_complete(async_main())
  loop.close()
```

I/O-bound: Sequential vs Asyncio

In [9]:
```
# Versi sekuensial untuk perbandingan
!python3 src/python/hello_world_io.py
```

```
Finished all jobs in 588.41 ms
```

In [14]:
```
# Versi asyncio
!python3 src/python/hello_world_io_asyncio.py
```

```
Finished all jobs in 578.92 ms
```

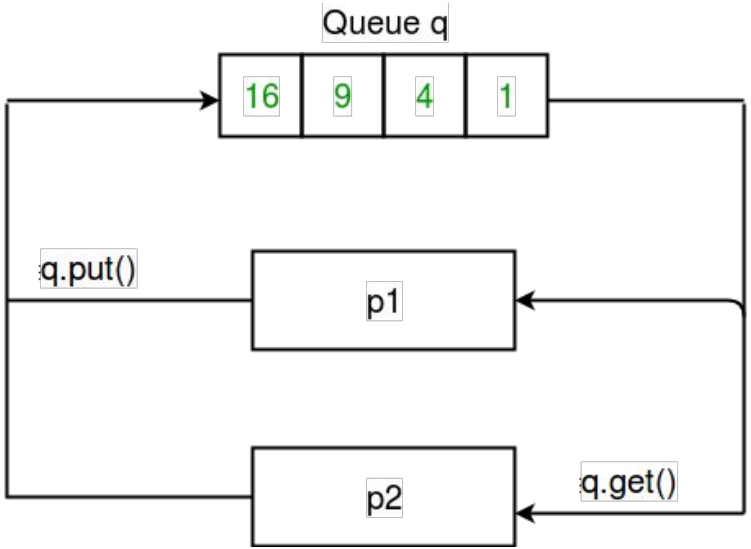
Multiprocessing in **(C)Python**

Multiprocessing in (C)Python

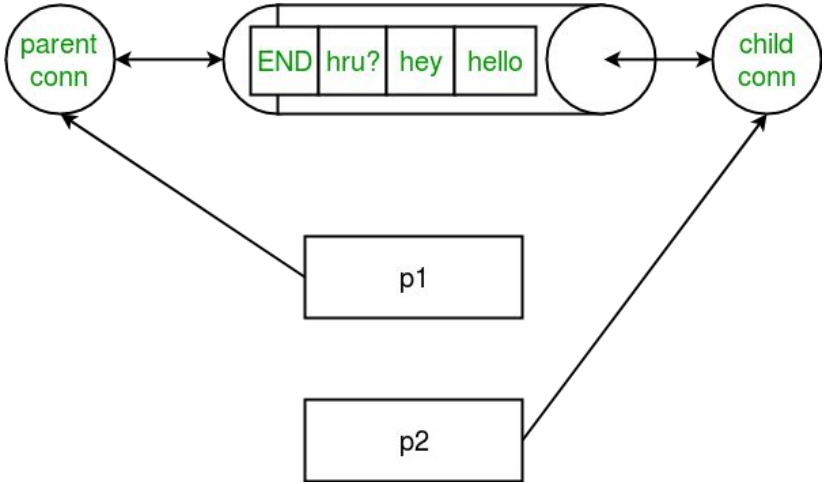
- *Built-in package* untuk *parallelism*
- Menggunakan banyak *core* CPU; masing-masing *process* 1 *core* CPU
- Interpreter sendiri-sendiri masing-masing *process*; ruang memori terpisah
- Pertukaran objek dan sinkronisasi *state* pada *multiprocess* tidak semudah pada *single process*.

Komunikasi antar *process*

multiprocessing.Queue



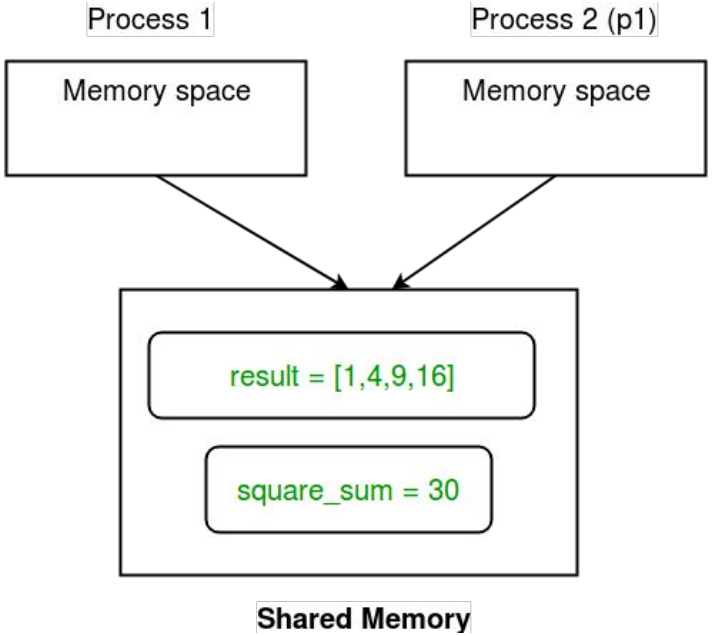
multiprocessing.Pipe



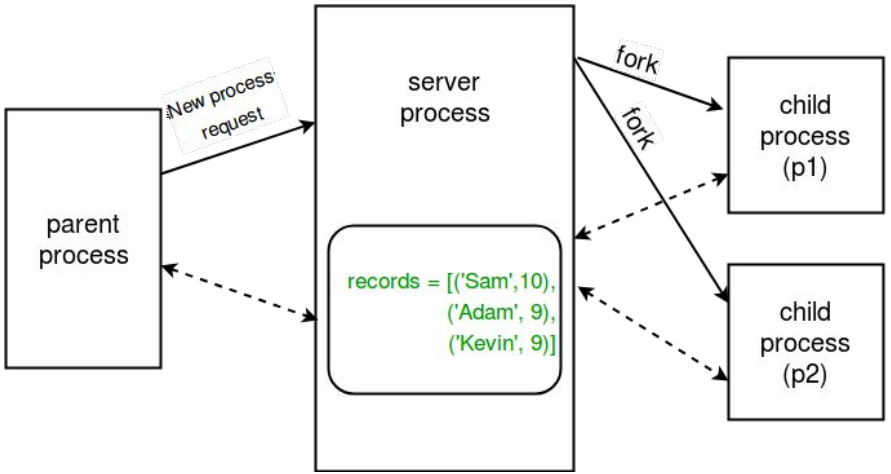
Sumber gambar: https://www.geeksforgeeks.org/multiprocessing-python-set-2/

Sinkronisasi *state* antar *process*

Shared memory



Server *process*



Sumber gambar: https://www.geeksforgeeks.org/multiprocessing-python-set-2/

Class inheritance

```
import multiprocessing as mp

class ProcessKu(mp.Process):
  def __init__(self, *args, **kwargs):
    mp.Process.__init__(self)
    ...

  def run(self):
    ...

if __name__ == "__main__":
  ...
  p_0 = ProcessKu(*args, **kwargs)
  p_0.start()
  ...
  p_0.join()
  ...
```

Process pool

```
import multiprocessing as mp

def function_ku(*args, **kwargs):
  ...

if __name__ == "__main__":
  ...
  with mp.Pool() as pool:
    ...
    pool_fut = pool.apply_async(
      function_ku, args, kwargs)
    ...
    pool.map(function_ku, iterable)
    ...
```

CPU-bound: Sequential vs Multiprocessing

In [2]:
```
# Versi sekuensial untuk perbandingan
!python3 src/python/hello_world.py
```

```
Finished with 9.854k operations, about 9.854k operation per second
```

In [16]:
```
# Berikut dapat dilihat cara multiprocessing yang pertama
!python3 src/python/hello_world_mp.py
```

```
Finished all jobs, totalling 37.114k operations, about 36.810k operations per second
```

In [18]:
```
# Berikut dapat dilihat cara multiprocessing yang kedua
!python3 src/python/hello_world_mp_pool.py
```

```
Finished all jobs, totalling 37.16k operations, about 36.863k operations per second
```

I/O-bound: Sequential vs Multiprocessing

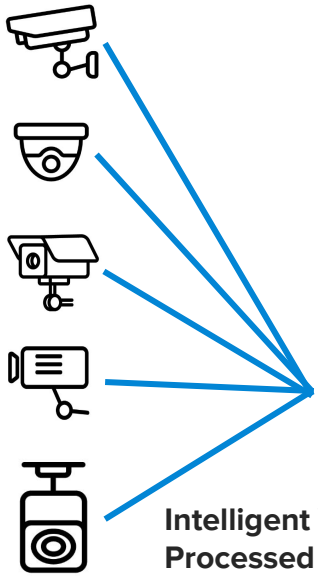
In [9]:
```
# Versi sekuensial untuk perbandingan
!python3 src/python/hello_world_io.py
```

```
Finished all jobs in 588.41 ms
```

In [20]:
```
# Multiprocessing untuk I/O
!python3 src/python/hello_world_io_mp_pool.py
```

```
Finished all jobs in 955.55 ms
```

Implementation



qluevision

Intelligent Video Analytics
Processed Parallelly

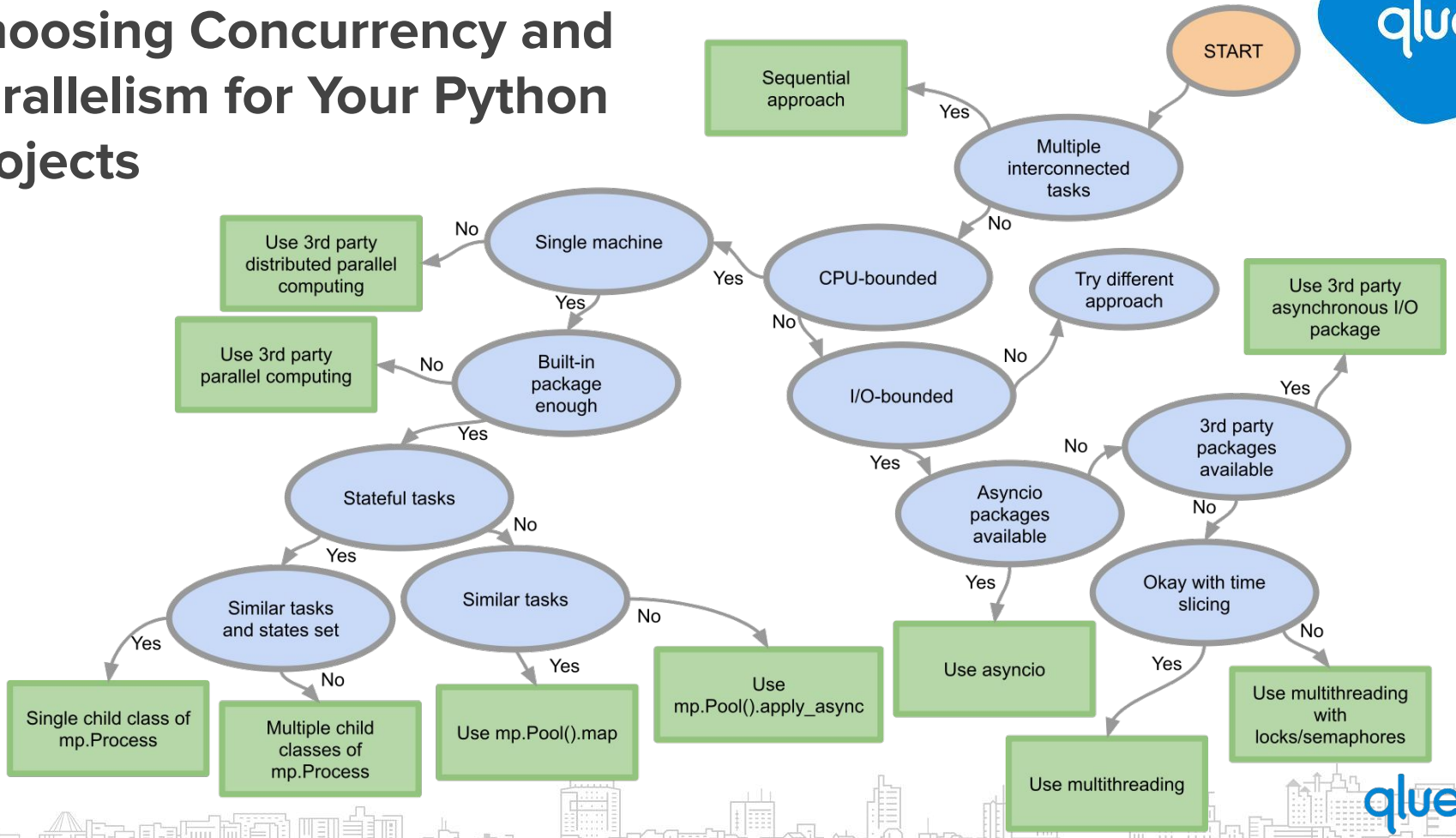
Web APIs
Served Concurrently

Summary

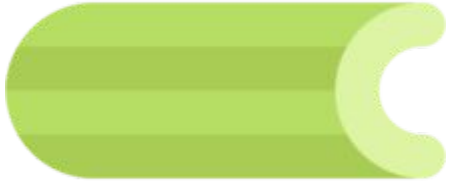
Summary

- asyncio (*cooperative multitasking*) untuk I/O-*bound* lebih baik daripada versi *sequential*
- threading (*pre-emptive multitasking*) untuk CPU-*bound* maupun I/O-*bound* tidak lebih cepat daripada versi sequential, namun tetap bisa digunakan jika tetap diinginkan concurrent I/O dan tidak tersedia packages versi asyncio
- multiprocessing (*the true parallelism*) untuk CPU-*bound* lebih cepat daripada versi *sequential*, namun tidak untuk I/O-bound

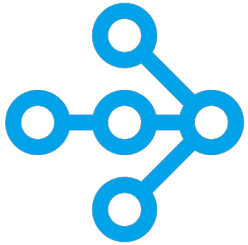
Choosing Concurrency and Parallelism for Your Python Projects



Concurrency and Parallelism Packages



gevent



RAY

References

https://hackernoon.com/why-concurrency-is-hard-a45295e96114
https://stackoverflow.com/questions/1050222/what-is-the-difference-between-concurrency-and-parallelism
https://realpython.com/python-concurrency/
https://docs.python-guide.org/starting/which-python/
https://www.jython.org/jythonbook/en/1.0/Concurrency.html
https://hackernoon.com/concurrent-programming-in-python-is-not-what-you-think-it-is-b6439c3f3e6a
https://whatis.techtarget.com/definition/preemptive-multitasking
https://www.pcmag.com/encyclopedia/term/48051/non-preemptive-multitasking
https://stackoverflow.com/questions/3042717/what-is-the-difference-between-a-thread-process-task
https://www.hellsoft.se/understanding-cpu-and-i-o-bound-for-asynchronous-operations/
https://realpython.com/python-gil/
https://docs.python.org/3.6/library/threading.html
https://docs.python.org/3.6/library/multiprocessing.html
https://docs.python.org/3.6/library/asyncio.html
https://docs.python.org/3.6/library/queue.html

Link to the Jupyter Notebook:

tinyurl.com/concurrency-pyconid2019

Follow us on **social media!**

Qlue Smart City

@qluesmartcity

Qlue Smart City

@qluesmartcity