11 Scaling and architecture

Nowadays all the hype is about resiliency and scalability, so I assume this is something that your development process is going to have to take into account sooner or later. Many sides of the issue are not particularly tied to Python itself, while some are only relevant to its main implementation, CPython.

The scalability, concurrency and parallelism of an application largely depend on the choices made about its initial architecture and design. As you'll see, there are some paradigms – like multi-threading – that don't apply correctly to Python, whereas other techniques, such as service oriented architecture, work better.

11.1 A note on multi-threading

What is multi-threading? It's the ability to run code on separate processors [1] inside a single Python process. This means that different parts of your code will be run in parallel.

Why is this needed? The most common cases are:

1. You need to run background tasks without stopping your main thread's execution, e.g. in the case of a graphical user interface where the main loop is waiting for events.

[1]Or sequentially on one, if multiple CPUs aren't present

2. You need to spread your work-load across several CPUs.

So at first, it may seem that multi-threading looks like a good way to scale and parallelize your application, solving these problems. When you want to spread a workload, you start a new thread for each new request instead of handling them one at a time.

Wonderful. Job done. We can move on.

No – sorry! First, if you've been in the Python world for a long time, you've probably encountered the word *GIL*, and know how hated it is. The GIL is the Python Global Interpreter Lock, a lock that must be acquired each time *CPython* [2] needs to execute byte-code. Unfortunately, this means that if you try to scale your application by making it run multiple threads, you'll always be limited by this global lock.

So while using threads seems like the ideal solution, in fact most applications I've seen running requests in multiple threads struggle to attain 150% CPU usage – i.e. 1.5 cores used. With computing nodes nowadays not usually having less than 2 or 4 cores, it's a shame. Blame the GIL.

There isn't currently any work being done to remove the GIL in *CPython*, because nobody thinks the solution is worth the difficulty of implementing and maintaining it.

However, *CPython* is just one [3] of the available Python implementations. Jython, for example, doesn't have a global interpreter lock, which means that it can run multiple threads in parallel efficiently. Unfortunately, these projects by their very nature lag behind *CPython*, and so are not really useful targets.

[2]The reference implementation of Python written in C that you run by typing *python* in your shell.
[3]although the most commonly used.

Note

PyPy is another Python implementation, but is written in Python (see Section 10.6). *PyPy* has a GIL too, but very interesting work is happening right now to replace it with a STM (Software Transactional Memory)-based implementation. This is something very exciting that's going to change how we build and run multi-threading software in the future. Hardware support is starting to appear in some processors, and Linux kernel developers are looking at ways to suppress kernel locks too. These are good signs.

So are we back to our initial use cases, with no good solutions on offer? Not true – there's (at least) two solutions you can use:

1. If you need to run background tasks, the easiest way to do that is to build your application around an event loop. There's a lot of different Python modules which provide for this, even a standard one called `asyncore`, which is an effort to standardize this functionality as part of PEP 3156. Some frameworks such as Twisted are built around this concept. The most advanced ones should give you access to events based on signals, timers and file descriptors activity – we'll talk about this in Section 11.3.
2. If you need to spread the work-load, using multiple processes is going to be more efficient and easier. See Section 11.2.

For us developers, mere mortals, it all means that we should think twice before using multi-threading. I've used multi-threading to dispatch jobs in rebuildd, a Debian build daemon I wrote a few years ago. While it seemed handy to have a thread to control each running build job, I very quickly fell into the concurrency trap. If I had the chance to begin again, I'd build something based on asynchronous events handling or multi-processing, and not have to worry about this problem.

Getting multi-threaded applications right is hard. The level of complexity means that it is a larger source of bugs than most others – and considering the little to be gained generally, it's better not to waste too much effort on it.

11.2 Multiprocessing vs multithreading

As explained earlier, multi-threading is not a good scalability solution because of the *GIL*. A better solution is the **multiprocessing** package that is provided with Python. It provides the same kind of interface that you would have using the **multithreading** module, except that it starts new processes (via *fork(2)*) rather than new system threads.

The below program is a simple example, which sums one million random integers 8 times, spread across 8 threads at the same time.

Worker using multithreading

```python
import random
import threading

results = []

def compute():
    results.append(sum(
        [random.randint(1, 100) for i in range(1000000)]))

workers = [threading.Thread(target=compute) for x in range(8)]
for worker in workers:
    worker.start()
for worker in workers:
    worker.join()
print("Results: %s" % results)
```

Running this program returns the following:

Example 11.1 Result of *time python worker.py*

```
$ time python worker.py
Results: [50517927, 50496846, 50494093, 50503078, 50512047, ↩
   50482863, 50543387, 50511493]
python worker.py  13.04s user 2.11s system 129% cpu 11.662 total
```

This has been run on an idle 4 cores CPU, which means that Python could have used up to 400% CPU power. But it was clearly unable to do that, even with 8 threads running in parallel – it stuck at 129%, which is just 32% of the hardware's capabilities.

Now, let's rewrite this implementation using **multiprocessing**. For a simple case like this, it's pretty straightforward:

Example 11.2 Worker using multiprocessing

```python
import multiprocessing
import random


def compute(n):
    return sum(
        [random.randint(1, 100) for i in range(1000000)])


# Start 8 workers
pool = multiprocessing.Pool(8)
print("Results: %s" % pool.map(compute, range(8)))
```

Running this program under the exact same conditions gives the following result:

Example 11.3 Result of *time python worker.py*

```
$ time python workermp.py
Results: [50495989, 50566997, 50474532, 50531418, 50522470, ↩
   50488087, 50498016, 50537899]
```

```
python workermp.py  16.53s user 0.12s system 363% cpu 4.581 total
```

The execution time has been reduced by 60%; this time, we have been able to consume up to 363% of CPU power, which is more than 90% of the computer's CPU capacity.

A further note – the **multithreading** module is not only able to efficiently spread a work-loads over several local processors, but can also do so over a network, through its **multithreading.managers** objects. It also provides bi-directional communication transports so your processes can exchange information with each other.

Each time you think that you can **parallelize** some work for a certain amount of time, it's much better to rely on multi-processing and to fork your jobs, in order to spread the workload among several CPU cores.

11.3 Asynchronous and event-driven architecture

Event-driven programming is a good solution to organize program flow in a way which listens for various events at once, without using a multi-threaded approach.

Consider an application that wants to listen for connection on a socket and then process the connection it receives. There are basically three ways to approach the problem:

1. Fork a new process each time a new connection is established, relying on something like the *multiprocessing* module.
2. Start a new thread each time a new connection is established, relying on something like the *threading* module.
3. Add this new connection to your event loop, and react to the event it will generate when it occurs.

It is (now) well known that listening to hundreds of event sources is going to scale much better when using an event-driven approach than, say, a thread-per-event approach [4]. This doesn't mean that the two techniques are not compatible, but it does mean that you can usually get rid of multiple threads by using an event-driven mechanism.

We've already covered the pros and cons of the first options; in this section, only the event-driven mechanism will be discussed.

The technique behind event-driven architecture is the building of an event loop. Your program calls a function that blocks until an event is received. The idea behind this is that your program can be kept busy while waiting for inputs and outputs to complete; the most basic events are "I have data ready to be read" or "I can now write data without blocking".

In Unix, the standard functions used to build such an event loop are the system calls `select(2)` or `poll(2)`. They expect a few file descriptors to listen for, and will react when one of them is ready to be read from or written to.

In Python, these system calls are exposed through the `select` module. It's easy enough to build an event-driven system with them, though it can be tedious.

Example 11.4 Basic example of using `select`

```
import select
import socket

server = socket.socket(socket.AF_INET,
                       socket.SOCK_STREAM)
# Never block on read/write operations
server.setblocking(0)

# Bind the socket to the port
```

[4] For further reading on this, take a look at the C10K problem.

```
server.bind(('localhost', 10000))
server.listen(8)

while True:
    # select() returns 3 arrays containing the object (sockets, files…)  ↩
       that
    # are ready to be read, written to or raised an error
    inputs, outputs, excepts = select.select(
        [server], [], [server])
    if server in inputs:
        connection, client_address = server.accept()
        connection.send("hello!\n")
```

A wrapper around these low-level interfaces was added to Python in the early days, called `asyncore`. It is not widely used, and hasn't evolved much.

Alternatively, there are many frameworks which provide this kind of functionality in a more integrated manner, such as Twisted or Tornado. Twisted has been almost a de-facto standard for years in this regard. C libraries that export Python interfaces, such as libevent, libev or libuv, also provides very efficient event loops.

While they all solve the same problem, the downside is that nowadays there are too many choices, and most of them are not interoperable. Also, most of them are callback based – which means that the program flow is not really clear when reading the code.

What about gevent or Greenlet? They avoid the use of callback, but the implementation details are scary, and include CPython x86 specific code and monkey-patching of standard functions. Not something you want to use and maintain on the long term, really.

Recently, Guido Van Rossum started to work on a solution code-named *tulip*, which is documented under PEP 3156.[5] The goal of this package is to provide a standard event loop interface. In the future, all frameworks and libraries would be compatible with it and would be able to interoperate.

tulip has been renamed and merged into Python 3.4 as the *asyncio* package. If you don't plan to depend on Python 3.4, it's also possible to install it for Python 3.3 using the version provided on PyPI – simply running `pip install asyncio` will do the job. Victor Stinner started a backport of *tulip* named *trollius*, which aims to be compatible with Python 2.6 and superior versions.

Now that you've got all the cards in your hand, no doubt you're wondering: but **what should I use to build an event loop in my event-driven application?**

At this point in Python's development, it's a really tough question. The language is still in a transition phase. As of the time of this writing, nothing yet uses the `asyncio` module. That means that using is going to be a real challenge.

Here are my recommendations at this point:

- If you target Python 2 only, `asyncio` is out of reach for you. For me, the next best choice would be something based on `libev`, like pyev.
- If you target both major Python versions – 2 and 3 – you'd better use something that is compatible with both, such as pyev. However, I would strongly advise you to keep in mind that you might have to transition later to `asyncio`. It may be useful to have a minimal abstraction layer, and not to spread the internal guts of your eventing-dependency over the entire program. If you're adventurous, trying to mix `asyncio/trollius` can be a nice solution too.
- If you only target version 3, go ahead with `asyncio`. It'll be a pain to start with, as there are still not a lot of examples or documentation, but it's a safe bet. You'll be a pioneer.

[5]*Asynchronous IO Support Rebooted: the "asyncio" Module*, Guido van Rossum, 2012

Example 11.5 Example with pyev

```
import pyev
import socket

server = socket.socket(socket.AF_INET,
                       socket.SOCK_STREAM)
# Never block on read/write operations
server.setblocking(0)

# Bind the socket to the port
server.bind(('localhost', 10000))
server.listen(8)

def server_activity(watcher, revents):
    connection, client_address = server.accept()
    connection.send("hello!\n")
    connection.close()

loop = pyev.default_loop()
watcher = pyev.Io(server, pyev.EV_READ, loop, server_activity)
watcher.start()
loop.start()
```

As you can see here, the `pyev` interface is pretty easy to grasp. Via its `libev` usage, it supports an `Io` object for input/output, but also the tracking of child processes, timers, signals and even callbacks to call when idle. `libev` also automatically relies on the best interface for polling – `epoll(2)` on Linux or `kqueue(2)` on BSD.

11.4 Service-oriented architecture

Considering the previously stated problems and solutions, the shortcomings of Python in terms of scalability and usage in large, complex applications can seem tricky to circumvent. However it appears that Python is really good at implementing Service-Oriented Architecture (SOA) – if you're not yet familiar with this, there's plenty of documentation and opinions that you can read online.

SOA is the architecture type used by OpenStack in all its components. Components use HTTP REST to communicate with external clients (end-users) and an abstracted RPC mechanism that can support several wire protocols, the most commonly used one being AMQP.

In your own case, the choice of which communication channels to use between those blocks is mainly a matter of knowing with whom you will be communicating.

When exposing an API to the outside world, the preferred channel nowadays is HTTP, and especially stateless designs such as REST [6] style architectures. These kinds of architectures are easy to implement, scale, deploy and comprehend.

However, when exposing and using your API internally, using HTTP may be not the best protocol. A large panel of communication protocols for applications exist, and a full description of any of them would likely fill an entire book.

In Python, there's plenty of libraries to build RPC [7] systems. Kombu – among others – is interesting because it provides an RPC mechanism on top of a lot of back-ends; AMQ protocol being the main one. But support for Redis, MongoDB, BeanStalk, Amazon SQS, CouchDB, or ZooKeeper are also provided.

In the end, there's a huge amount to be gained indirectly from using such loosely coupled architecture. If we consider that each module provides and exposes an API,

[6] Representational state transfer
[7] Remote Procedure Call

we can run multiple daemons exposing this API. For example, *Apache httpd* would create a new worker using a new system process that handles new connections; we can then dispatch our connection to a different worker running on the same compute node. All we need to have is a system of dispatching the work between our workers, which provides this API. Each block will be a different Python process, and as we've seen above, this is better than multi-threading to spread your work-load. You'll be able to start multiple workers on each computing node you have. Even if not strictly necessary, using stateless blocks should be favored any time you have the choice.

ZeroMQ is a socket library that can act as a concurrency framework. The following example implements the same worker seen in the previous examples, but uses ZeroMQ as a way to dispatch and communicate.

Workers using ZeroMQ

```
import multiprocessing
import random
import zmq


def compute():
    return sum(
        [random.randint(1, 100) for i in range(1000000)])


def worker():
    context = zmq.Context()
    work_receiver = context.socket(zmq.PULL)
    work_receiver.connect("tcp://0.0.0.0:5555")
    result_sender = context.socket(zmq.PUSH)
    result_sender.connect("tcp://0.0.0.0:5556")
    poller = zmq.Poller()
    poller.register(work_receiver, zmq.POLLIN)
    while True:
        socks = dict(poller.poll())
        if socks.get(work_receiver) == zmq.POLLIN:
            obj = work_receiver.recv_pyobj()
            result_sender.send_pyobj(obj())

context = zmq.Context()
# Build a channel to send work to be done
work_sender = context.socket(zmq.PUSH)
work_sender.bind("tcp://0.0.0.0:5555")
# Build a channel to receive computed results
result_receiver = context.socket(zmq.PULL)
result_receiver.bind("tcp://0.0.0.0:5556")
# Start 8 workers
processes = []
for x in range(8):
    p = multiprocessing.Process(target=worker)
    p.start()
    processes.append(p)
# Start 8 jobs
for x in range(8):
    work_sender.send_pyobj(compute)
# Read 8 results
results = []
for x in range(8):
    results.append(result_receiver.recv_pyobj())
# Terminate all processes
for p in processes:
    p.terminate()
```

```
print("Results: %s" % results)
```

As you can see, ZeroMQ provides an easy way to build communication channels. I've chosen the TCP transport layer here to illustrate the fact that we could run this over a network. It should be noted that ZeroMQ also provides a *inproc* communication channel that works by using Unix sockets. Obviously the communication protocol built upon ZeroMQ in this example is very simplistic – in order to keep this book's examples clear and concise; but it shouldn't be hard to imagine building a more sophisticated communication layer on top of it.

With such a protocol, it's easy to imagine building a entirely distributed application communication with a network message bus – ZeroMQ, AMQP, or something else.

Note also that protocols like HTTP, ZeroMQ or AMQP are language agnostic; you can use different languages and platforms to implement each part of your system. While we all agree that Python is a good language, other teams might have other preferences; or another language might be a better solution for some part of a problem.

In the end, using a transport bus to decouple your application is a good option. It allows you to build both synchronous and asynchronous APIs that can be spread from one computer to several thousand. It doesn't tie you to a particular technology or language – and these days, there's no longer a reason not to be ready to distribute your software, or to be constrained by any particular language.