Ignore All Web Performance Benchmarks, Including This One (/post/ignore-all-web-performance-benchmarks-including-this-one)

September 15 2020

Posted by **Miguel Grinberg (/author/Miguel Grinberg)** under **Python (/category/Python)**.

A couple of months ago there was an article titled Async Python is Not Faster (http://calpaterson.com/async-python-is-not-faster.html) making the rounds on social media. In it, the author Cal Paterson made the point that contrary to popular belief, asynchronous web frameworks are not only "not faster" than their traditional synchronous counterparts, but they are also slower. He supports this by showing the results of a fairly complete benchmark that he implemented.

I wish things were as simple as this author puts them in his blog post, but the fact is that measuring web application performance is incredibly complex and he's got a few things wrong, both in the implementation of the benchmark and in his interpretation of the results.

In this article you can see the results of my effort in understanding and fixing this benchmark, re-running it, and finally arriving at a shocking revelation.

The Benchmark Results

Before I delve into the gory details, I assume you are anxious to see what the results of the benchmark are. These are the results that I obtained when running this benchmark, after I fixed all the problems I've found in it. I have also added several more frameworks that I was particularly interested in:

Framework	Web Server	Type	Wrk	Tput	P50	P99	#DB
Falcon	Meinheld	Async / Greenlet	6	1.60	94	163	100
Bottle	Meinheld	Async / Greenlet	6	1.58	96	167	100
Flask	Meinheld	Async / Greenlet	6	1.43	105	182	100
Aiohttp	Aiohttp	Async / Coroutine	6	1.38	108	196	91
Falcon	Gevent	Async / Greenlet	6	1.37	107	223	100
Sanic	Sanic	Async / Coroutine	6	1.34	111	220	85
Bottle	Gevent	Async / Greenlet	6	1.31	114	220	100
Starlette	Uvicorn	Async / Coroutine	6	1.31	113	236	87
Tornado	Tornado	Async / Coroutine	6	1.25	120	230	93
Flask	Gevent	Async / Greenlet	6	1.22	122	241	98
FastAPI	Uvicorn	Async / Coroutine	6	1.21	124	245	86
Sanic	Uvicorn	Async / Coroutine	6	1.18	127	251	87
Quart	Uvicorn	Async / Coroutine	6	1.13	132	252	79
Aioflask	Uvicorn	Async / Coroutine	6	1.11	135	269	80
Falcon	uWSGI	Sync	19	1.04	146	179	19
Bottle	uWSGI	Sync	19	1.03	148	184	19
Falcon	Gunicorn	Sync	19	1.02	149	190	19
Flask	uWSGI	Sync	19	1.01	151	182	19
Flask	Gunicorn	Sync	19	1.00	153	184	19
Bottle	Gunicorn	Sync	19	0.99	153	208	19
Quart	Hypercorn	Async / Coroutine	6	0.91	161	336	69

Note that the above results were generated during an update of this article on October 27th, 2020 that addressed an issue with the P99 numbers. In the spirit of full transparency, you can click here to view the results that were published with the original article. Click here to restore the most up to date results.

Notes regarding these results:

- This benchmark shows performance under a constant load of one hundred clients.
- There are three types of tests: Sync, Async / Coroutine and Async / Greenlet. If you need to understand what the differences between these types are, check out my Sync vs. Async Python (https://blog.miguelgrinberg.com/drafts/eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MTY3fQ.3nDqIIIe7e3YS8C44T_tJPxAZZGQwEh w2GUBvCxQA) article.

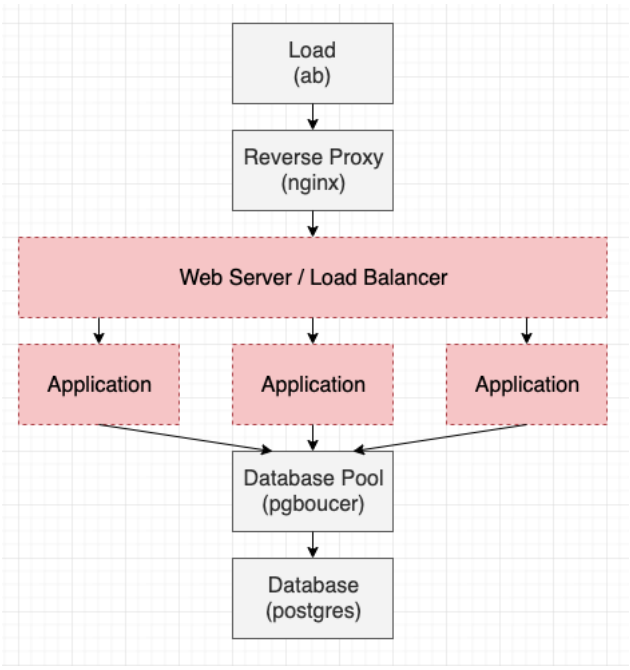
- I used two different worker configurations. For async tests I used 6 workers (one per CPU). For sync tests I used 19 workers. I arrived at these numbers by testing different configurations to maximize performance.
- All the asyncio tests use uvloop (https://github.com/MagicStack/uvloop) for best performance.
- Instead of reporting throughput as requests per second, I use the Flask+Gunicorn test as the baseline, and report the throughput for each test as a multiplier from this baseline. For example, a throughput of 2.0 means "twice as fast as Flask+Gunicorn" and a throughput of 0.5 means "half as fast (or twice as slow) as Flask+Gunicorn".
- P50 is the 50th percentile or median of the request processing time in milliseconds. In other words, 50% of the requests sent during the test were completed in less than this time.
- P99 is the 99th percentile of the request processing time in milliseconds. You can think of this number as a longest a request took to be processed, with the outliers removed.
- The #DB column shows the maximum number of database sessions used by each test. Per configuration there were 100 sessions available. Sync tests are obviously limited to one session per worker.

Note that you can sort the data in every possible way by clicking on the table headers. When you are done, keep on reading to understand what do these numbers really mean (and why none of this matters in practice!).

What Does The Benchmark Do?

The benchmark consists on running a web application under load, and measuring the performance. The test is repeated for many different configurations of web servers and web frameworks to determine how all these tools perform under the same conditions.

Below you can see a diagram of what the test does. In this diagram, the gray boxes are constant, while the red boxes represent the parts of the system in which the different implementations that are being evaluated are plugged in.



- The **load generator** is the process that generates client connections. This is done with Apache Bench (ab) (https://httpd.apache.org/docs/2.4/programs/ab.html).
- Requests are received by a **reverse proxy**, which is the only public interface. The nginx (https://www.nginx.com/) server provides this functionality.
- The **web server and load balancer** accepts requests from the reverse proxy and dispatches them to one of several web application workers.
- The **application** component is where requests are handled.
- The **database pool** is a service that manages a pool of database connections. In this test this task is done by pgbouncer (https://www.pgbouncer.org/).
- The **database** is the actual storage service, which is a PostgresSQL (https://www.postgresql.org/) instance.

The original benchmark had a nice variety of web servers. I've added a couple more that were interesting to me. The complete list of web servers that I tested is shown below.

Server	Type	Language
Gunicorn (https://gunicorn.org/)	Sync	Python

Server	Type	Language
uWSGI (https://uwsgi-docs.readthedocs.io/en/latest/)	Sync	C
Gevent (http://www.gevent.org/)	Async / Greenlet	Python
Meinheld (https://meinheld.org/)	Async / Greenlet	C
Tornado (https://www.tornadoweb.org/en/stable/)	Async / Coroutine	Python
Uvicorn (https://www.uvicorn.org/)	Async / Coroutine	Python
Aiohttp (https://docs.aiohttp.org/en/stable/)	Async / Coroutine	Python
Sanic (https://sanic.readthedocs.io/en/latest/)	Async / Coroutine	Python
Hypercorn (https://pgjones.gitlab.io/hypercorn/)	Async / Coroutine	Python

For the application components, a small microservice that performs a database query and returns the result as a JSON response is used. So that you have a better idea of what the test involves, below you can see the Flask and Aiohttp implementations of this service:

```
import flask
import json
from sync_db import get_row

app = flask.Flask("python-web-perf")

@app.route("/test")
def test():
    a, b = get_row()
    return json.dumps({"a": str(a).zfill(10), "b": b})
```

```
import json
from aiohttp import web
from async_db import get_row

async def handle(request):
    a, b = await get_row()
    return web.Response(text=json.dumps({"a": str(a).zfill(10), "b": b}))

app = web.Application()
app.add_routes([web.get('/test', handle)])
```

The `get_row()` function runs a query on a database loaded with random data. There are two implementations of this function, one with the psycopg2 (https://www.psycopg.org/docs/) package for standard Python, and another one with aiopg (https://aiopg.readthedocs.io/en/stable/) for `asyncio` tests. For greenlet tests, `psycopg2` is properly patched so that it does not block the async loop (this was an important oversight in the original benchmark).

The implementations of this application that I tested are based on the following web frameworks:

Framework	Platform	Gateway interface
Flask (https://flask.palletsprojects.com/en/1.1.x/)	Standard Python	WSGI
Bottle (https://bottlepy.org/docs/dev/)	Standard Pyhon	WSGI
Falcon (https://falcon.readthedocs.io/en/stable/)	Standard Pyhon	WSGI
Aiohttp (https://docs.aiohttp.org/en/stable/)	asyncio	Custom
Sanic (https://sanic.readthedocs.io/en/latest/)	asyncio	Custom or ASGI
Quart (https://pgjones.gitlab.io/quart/)	asyncio	ASGI
Starlette (https://www.starlette.io/)	asyncio	ASGI
Tornado (https://www.tornadoweb.org/en/stable/)	asyncio	Custom
FastAPI (https://fastapi.tiangolo.com/)	asyncio	ASGI
Aioflask (https://github.com/miguelgrinberg/aioflask)	asyncio	ASGI

I have not tested every possible pairing of a web server and an application, mainly because some combinations do not work together, but also because I did not want to waste testing time on combinations that are weird, uncommon or not interesting.

If you are familiar with the original benchmark, here is the list of differences in my own set up:

- I have executed all the tests on real hardware. The original benchmark used a cloud server, which is not a good idea because CPU performance in virtualized servers is constantly changing and is dependent on usage by other servers that are colocated with yours on the same physical host.
- I have used Docker containers to host all the components in the test. This is just for convenience, I do not know what was the setup in the original benchmark.

- I have removed the session pool at the application layer, since `pgbouncer` already provides session pooling. This solves an indirect problem where the application pools for sync and async tests were configured differently. In my test there are 100 sessions available to be distributed among all the application workers.
- The database query issued in the original benchmark was a simple search by primary key. To make the test more realistic I made the query slightly slower by adding a short delay to it. I understand that this is an extremely subjective area and many will disagree with this change, but what I observed is that with such quick queries there wasn't much opportunity for concurrency.
- I mentioned above that I patched `psycopg2` to work asynchronously when used with a greenlet framework. This was omitted in the original benchmark.
- The `aiohttp` test in the original benchmark used the standard loop from `asyncio` instead of the one from `uvloop`.

What Do These Results Mean?

There are a few observations that can be made from the results that I obtained, but I encourage you to do your own analysis of the data and question everything. Unlike most benchmark authors, I do not have an agenda, I'm only interested in the truth. If you find any mistakes, please reach out and let me know.

I bet most of you are surprised that the best performing test managed a mere 60% performance increase over a standard Flask/Gunicorn deployment. There are certainly performance variations between the different servers and frameworks, but they are not that large, right? Remember this the next time you look at a too-good-to-be-true benchmark published by an asyncio framework author!

The async solutions (with the exception of the Hypercorn server which appears to be extremely slow) clearly perform better than the sync ones on this test. You can see that overall the sync tests are all towards the bottom of the list in throughput, and all very close to the Flask/Gunicorn baseline. Note that the author of the original benchmark refers to greenlet tests as sync for some strange reason, giving more importance to the coding style in which the application is written than to the method concurrency is achieved.

If you look at the original benchmark results (http://calpaterson.com/async-python-is-not-faster.html) and compare them against mine, you may think these are not coming from the same benchmark. While the results aren't a complete inverse, in the original results the sync tests fared much better than in mine. I believe the reason to be that the database query issued in the original benchmark was extremely simple, so simple that there was little or no gain in running multiple queries in parallel. This puts the async tests at a disadvantage, since async performs the best when the tasks are I/O bound and can overlap. As stated above, my version of this benchmark uses a slower query to make this a more realistic scenario.

One thing the two benchmarks have in common is that Meinheld tests did very well in both. Can you guess why? Meinheld is written in C, while every other async server is written in Python. That matters.

Gevent tests did reasonably well in my benchmark and terribly in the original one. This is because the author forgot to patch the psycopg2 package (https://github.com/psycopg/psycogreen/) so that it becomes non-blocking under greenlets.

Conversely, uWSGI tests did well in the original benchmark, and just average in mine. This is odd, since uWSGI is also a C server, so it should have done better. I believe using longer database queries has a direct effect on this. When the application does more work, the time used by the web server has less overall influence. In the case of async tests, a C server such as Meinheld matters a lot because it uses its own the loop and performs all the context-switching work. In a sync server, where the OS does the context-switching, there is less expensive work that can be optimized in C.

The P50 and P99 numbers are much higher in my results, in part because my test system is probably slower, but also because the database queries that I'm issuing take longer to complete and this translates into longer request handling times. The original benchmark just queried a row by its primary key, which is extremely fast and not at all representative of a real-world database use.

Some other conclusions I make from looking at my own benchmark:

- Looking at the three sync frameworks, Falcon and Bottle appear to be slightly faster than Flask, but really not by a margin large enough to warrant switching, in my opinion.
- Greenlets are awesome! Not only they have the best performing asynchronous web server, but they also allow you to write your code in standard Python using familiar frameworks such as Flask, Django, Bottle, etc.
- I'm thrilled to find that my Aioflask experiment (https://github.com/miguelgrinberg/aioflask) performs better than standard Flask, and that it is in the same performance level as Quart. I guess I'm going to have to finish it.

Benchmarks Are Unreliable

I thought it was curious that due to a few bugs and errors of interpretation, this benchmark convinced the original author that sync Python is faster than async. I wonder if he already had this belief before creating the benchmark, and if that belief is what led him to make these unintentional mistakes that veered the benchmark results in the direction he wanted.

If we accept that this is possible, shouldn't we be worried that this is happening to me as well? Could I have been fixing this benchmark not for correctness and accuracy, but just so that it agrees more with my views than with his? Some of the fixes that I've made are really bugs. For example, not patching `psycopg2` when using greenlet servers cannot be defended as a choice, that is a clear cut bug that even the benchmark author could not justify (https://github.com/calpaterson/python-web-perf/issues/11). But what about other changes I've made that are more in a gray area, like how long the database query should take?

As a fun exercise, I decided to see if I could reconfigure this benchmark to show completely different results, while obviously preserving its correctness. The options that I had to play with are summarized in the following table, where you can see the configuration used in the original and my own version of the benchmark, along with the changes I would make if I wanted to tip the scale towards async or towards sync:

Option	Original	My Benchmark	Better Async	Better Sync
Workers	Variable	Sync: 19 Async: 6	Sync: 19 Async: 6	Sync: 19 Async: 6
Max database sessions	4 per worker	100 total	100 total	19 total
Database query delay	None	20ms	40ms	10ms
Client connections	100	100	400	19

Let me explain how changes to these four configuration variables would affect the tests:

- I decided to keep the number of workers the same, because by experimentation I have determined that these numbers were the best for performance on my test system. To me it would feel dishonest if I changed these numbers to less optimal values.
- Sync tests use one database session per worker, so any amount of sessions that is equal to or higher than the worker count results in similar performance. For async tests more sessions allow for more requests to issue their queries in parallel, so reducing this number is a sure way to affect their performance.
- The amount of I/O performed by the requests determine the balance between the I/O and CPU bound characteristics of the benchmark. When there is more I/O, async servers can still have good CPU utilization due to their high concurrency. For sync servers, on the other side, slow I/O means requests have to wait in a queue for longer until workers free up.
- Async tests can scale freely to large number of concurrent tasks, while sync tests have a fixed concurrency that is determined by the number of workers. Higher numbers of client connections hurt sync servers much more than async, so this is an easy way to favor one or the other.

Are you ready to be amazed? Below you can see a table that compares the throughput results I shared at the beginning of this article to the numbers I obtained by reconfiguring the benchmark for the two scenarios I just discussed. Click on the table headers to sort.

Framework	Web Server	Type	My Results	Better Async	Better Sync
Falcon	Meinheld	Async / Greenlet	1.60	5.17	1.12
Bottle	Meinheld	Async / Greenlet	1.58	5.56	1.13
Flask	Meinheld	Async / Greenlet	1.43	5.27	1.06
Aiohttp	Aiohttp	Async / Coroutine	1.38	4.79	1.27
Falcon	Gevent	Async / Greenlet	1.37	4.66	0.99
Sanic	Sanic	Async / Coroutine	1.34	4.58	1.09
Bottle	Gevent	Async / Greenlet	1.31	4.59	1.18
Starlette	Uvicorn	Async / Coroutine	1.31	4.36	1.13
Tornado	Tornado	Async / Coroutine	1.25	4.19	1.03
Flask	Gevent	Async / Greenlet	1.22	4.54	1.01
FastAPI	Uvicorn	Async / Coroutine	1.21	4.33	1.02
Sanic	Uvicorn	Async / Coroutine	1.18	4.40	1.03
Quart	Uvicorn	Async / Coroutine	1.13	3.99	0.99
Aioflask	Uvicorn	Async / Coroutine	1.11	3.57	1.07
Falcon	uWSGI	Sync	1.04	1.00	1.43
Bottle	uWSGI	Sync	1.03	0.90	1.35
Falcon	Gunicorn	Sync	1.02	1.00	1.11
Flask	uWSGI	Sync	1.09	1.01	1.26
Flask	Gunicorn	Sync	1.00	1.00	1.00
Bottle	Gunicorn	Sync	1.04	0.99	1.11

Framework	Web Server	Type	My Results	Better Async	Better Sync
Quart	Hypercorn	Async / Coroutine	0.91	3.24	0.80

Isn't this mindblowing? Remember that the benchmark is always the same, all I'm doing is changing configuration parameters.

The "Better Async" benchmark shows that all sync tests are near the 1.0 baseline of the Flask/Gunicorn test, while the async tests are 3x to 6x times faster. Even the Hypercorn test, which was very slow in my benchmark, scored a very decent grade. The "Better Sync" benchmark shows the uWSGI tests doing better than the rest, and while most of the async tests ended up above 1.0, looking at these results would not excite anyone into going async.

Conclusion

I hope this article helped you realize now that the benchmark game is rigged. I could easily make reasonably sounding arguments in favor of any of these sets of results, which is exactly what every person releasing a benchmark does. I don't mean to say that benchmark authors are dishonest, in fact I believe most aren't. It's just that it is very difficult to set your personal views aside and be objective when constructing a benchmark and analyzing its results.

As I state in the title, I think the best advice I can give you is to understand the strengths async and sync solutions have and make a decision based on that instead of on what some benchmark says. Once you know which model works best for you, remember that the difference in performance between different frameworks or web servers isn't going to be very significant, so choose the tools that make you more productive!

If you are interested in playing with my version of this benchmark, you can find it on this GitHub repository (https://github.com/miguelgrinberg/python-web-perf).
