Analyzing the Performance of Python's Asynchronous Execution in a Server Herd

Abstract

It has come to our attention that while the Wikimedia Architecture and the LAMP infrastructure are both work fairly well, utilizing a complex network of web servers and routers to ensure reliable storage and data transfer, there lies a few potential drawbacks in its build. Particularly, the infrastructure fails if there are frequent updates to the main application server (much more queuing and infrastructure traffic), the clients are much more mobile (more updates to the server), and various protocols are to be used (like in LAMP, where adding a new server would be difficult due to the complexity of the infrastructure). So, we would be conducting a simple test by using Python to implement a network of simple web servers using ansyncio, and see how well it can perform.

1. Experiment Details

Generally, the experiment involves implementing 5 identical servers in a "server herd architecture" that communicates with clients, stores their location in memory, communicate with each other, share the clients' information between each other, and performs HTTP requests to Google using the Google Places API. In addition, each server is to work on a single GPU/core, and would work with multiple clients using asynchronous execution using asyncio.

To instantiate the servers, the Python program to run (server.py) would take a name parameter. The name parameter must match one of "Goloman", "Hands", "Holiday", "Wilkes", or "Welsh", and each name would map to a specific port number (here, 12070-12074). The program would then start an event loop and run forever, unless a KeyboardInterrupt were to happen, in which case the server would close. In addition, each server name is meant to communicate bidirectionally with other certain server names (Goloman talks with Hands, Holiday and Wilkes; Hands talks with Wilkes; Holiday talks with Welsh and Wilkes).

The server would take a series of requests from clients: "IAMAT" requests, "WHATSAT" requests, and "AT" messages (supposedly only from other peer servers). On an "IAMAT", the server would parse the request of the client name, and store the location (latitude, longitude), the difference in time between the particular server-client pair, and the time the record was made. The server would return an "AT" message to the same client, along with a copy of the original request made. The server would also propagate the record as a variation of an "AT" message to other servers using a flooding algorithm. In my implementation, the "AT" message is like the original "AT" response, but instead of concatenating the client's request, it would concatenate its own name and the name of the servers it is supposed to communicate with, so no other servers would try to communicate with them (no server is attempted to be accessed twice). On a "WHATSAT" request, the server is to follow the parameters of the request (client name, number of locations) communicate with a Google server using the Google Places API, and get a JSON response of the places within a radius of the specific client's location., and return the parsed JSON.

1.1. Analysis of Asyncio

The asyncio is the library is mainly used to have a program run multiple coroutines/functions concurrently. Generally, it works by having multiple functions run on the same CPU, but switch context and control whenever convenience to increase performance and decrease runtime. In a way, it can simply be described as efficient time-sharing of the CPU through the calling and awaiting of subroutines at different times. In Python, this is done through the "async" and "await" keywords.

In our case, our use of asyncio is meant to implement a server that can perform multiple potential actions, like communication using TCP, reading from a client, writing to a client, updating its own data, notify other peer servers, request data from Google using HTTP, and parsing the JSON response from the Google Places API. Of course, for our server, we are expected to run multiple of these action at the same time.

So, to implement this, we are to use the asyncio library to implement and event loop. Here, an event loop can be described as a construct that manages tasks to run. In other words, it is a semi-infinite loop that runs one task at a time, and if another waiting task can better utilize the time to run (say, when the current task is blocking from reading), then the event handler can pause the current task and run another for better time efficiency.

Through running an event loop, even though our server is single-CPU, we are now able to perform the timesharing of multiple threads on the same CPU/core to run at the same time.

1.2. Problems

Of course, because asyncio programs are meant to run asynchronously, we have little control over the order of execution of the variations and combinations of tasks to run. For instance, in our Python server program, if we were to have a task block from reading an "IAMAT" request, and then work with a subsequent "WHATSAT" request on the same client, we cannot guarantee that the update will

happen before evaluating the "WHATSAT" request. However, for the general case of shared resources, there are locks in the the asyncio library, but they don't apply in the case of our project. So, asyncio shares the same resolvable issues of race conditions as multithreading in general on multiple CPUs/cores.

In addition, asyncio are meant only for asynchronous execution, so it is not meant for the case of multiprocessing and taking advantage of multiple CPUs/cores. Multithreading on multiple CPUs/cores is outside the scope of asyncio (of course, it is possible to use asyncio with multiprocessinging, given compatible coroutines are running in the same processing). Also, note that Python has some issues with multithreading in general (more on that later).

1.3 Pro/Con

From what we see above, we can say that Python is definitely viable to write a server with in the setting of using a single CPU/core. However, from what we see, since Python's asyncio is to basically act as multithreading in a single CPU/core environment, it would not be making use of all resources in a multi-CPU/core system. However, considering that both methods can be combined, asyncio is a legitimate way to augment performance, no matter single- or multi-processing. In our server-herd setting, asyncio is very suitable, as it would be very scalable for this very reason, as we can add servers of difference scales without too much worry.

2. Comparison with Java-based Approach

If we are to compare our Python-based approach with a Java-based approach, we need to see the specifics of Python's performance in comparison to that of Java. In general, Java's performance is better than that of Python.

2.1. Type Checking

For type checking, we must first see how variables and types are handled in both languages. In the case of Java, the language can be called statically typed, in which the programmer must declare each variable's type. On the other hand, Python is dynamically typed, in which any object or primitive can be bound to any variable.

However, in Python, dynamic typing forces the interpreter to undergo type inference for each variable during runtime. This of course would be a performance hit, as time and computing power needs to be spent on this trivial task. In comparison to Java, in which the compiler can check the variable's type during compile time, there would be no type problems during run time as they would have already been caught, unlike in Python.

In addition, readability is also a factor as well. With Java's static typing, not just the compiler and the interpreter but the programmer can make out types much more easily. In Python, on the other hand, a variable can be of any type depending on what was assigned to it earlier, forcing the reader to put in more effort to understand the variables' typing in the code, reducing readability.

2.2. Memory Management

For memory management, Python is definitely more efficient with memory management mainly due to its use of reference counts. In reference counting, when an object is made, Python keeps track of all variables in the global scope that references the object (a variable can gain a reference through assignment), and the garbage collection would occur only when the reference count reaches zero. Of course, there would be a performance penalty with each assignment to update the counters, but it is more memory efficient in that only a single copy of a particular object would be created.

For Java, objects created in the heap would follow a mark-sweep-compact garbage collection system. The garbage collection would work concurrently with the program, and during garbage collection, it would mark-and-sweep the unused objects in the heap, whilc performing fragmentation on the objects that are in use, which would increase performance due to caching.

2.3 Multithreading

As mentioned earlier, asyncio is mainly meant for implementing concurrent execution. It is not easy to compare such an approach with multithreading in general, as multithreading is meant to make use of multiple cores. If we are to compare asyncio with multithreading in general, it would be that it would be more efficient in the case of a single CPU/core, but loses out on making use of multiple CPUs/cores.

However, for Python multithreading in general, only concurrent execution is supported, not parallel, and some implementations that don't support it very well at all, like CPython. CPython is implemented with a GIL (global interpreter lock, in which a thread can run only if it obtains the object), meaning that only one thread can run at a time, effectively banning any implementation of parallelism. Of course, one can bypass this with multiprocessing, but of course, inter-process communication wouldn't be as efficient as in the case if multithreading were allowed. Another solution would be to us another implementation like IronPython, but in general, we can say that support for parallel multithreading isn't too scalable or supported as opposed to Java.

3. Brief Comparison of Asyncio to Node.js

Finally, we now compare Python's asyncio with Javascript's Node.js. Like Python, there is little to no support for

multithreading in parallel in Javascript on multiple CPUs/cores. Both are written to allow for concurrent execution on a single CPU/core, but there are few fundamental differences in how concurrency is implemented in each, along with a few language differences that make working with each so different.

Let's begin with the design that implements concurrency in each.In both cases, the coroutines are run through an event loop, but for Python, the coroutines are managed by "futures", while in Javascript, they are managed by "promises". The difference between the two is that "futures" (abstractions of not-yet-evaluated values) are evaluated externally by external callbacks, while "promises" are evaluated internally. In a way, because of such a dependency on callbacks, context-switches are necessary, thus a hit on performance for Python.

In addition, Javascript supports 4 methods of concurrent execution in code: async/await (like in Python), callbacks (also supported in Python), and promise chaining.

For typing, both languages are dynamically typed, but Javascript is weakly typed, while Python is strongly typed. This means that because any object can be treated as another type (unlike in Python, in which one object can only be treated as a single type), Javascript is more prone to errors (although there would be less type checking, which is a performance benefit). In addition, as far coding style goes, Javascript would be less readable with weak typing, meaning it may be less scalable.

For memory management, Javascript is exactly like Python: it uses both mark-and-sweep and reference counting. As far as performance and memory efficiency goes, it is difficult to gauge.

4. Conclusion

All in all, from what we can see above, Python is suitable for the task of writing a server herd, but there do exist other alternatives that may perform better than Python. For performance, Java and Javascript can definite run faster, but they may not be as scalable (or in Javascript's case, as stable and error-avoiding).

References

[1] Bene, Adam. "Async/await vs Coroutines vs Promises vs Callbacks". Medium. https://blog.benestudio.co/async-await-vs-coroutines-vs-promises-eaedee4e0829

[2] "Differences between Futures in Python3 and Promises in ES6". *Stack Overflow*. 26 Nov, 2017. https://stackoverflow.com/questions/47493369/differences-between-futures-in-python3-and-promises-in-es6/47499994

[3] "GlobalInterpreterLock." *Python*. 2 August, 2017. https://wiki.python.org/moin/GlobalInterpreterLock

[4] Gupta, Lokesh. "Java Garbage Colletion Algorithms [till Java 9]." *HowToDoInJava*. 2016. https://howtodoinjava.com/java/garbage-collection/all-garbage-collection-algorithms/

[5] "Memory Management". *MDN Web Docs*. 19 Nov, 2018. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_Management

[6] "Memory Management in Various Languages." *Memory Management Reference*. 2018. https://www.memorymanagement.org/mmref/lang.html

[7] RadeCliffe, Tom. "Python Vs. Java: Duck Typing, Parsing On Whitespace And Other Cool Diferences." *ActiveState*. https://www.activestate.com/blog/python-vs-java-duck-typing-parsing-whitespace-and-other-cool-differences/

[8] Saba, Sahand. "Understanding Asynchronous IO With Python's Asyncio And Node.js." *Math U Code*. 10 Oct, 2014. https://sahandsaba.com/understanding-asyncio-node-js-python-3-4.html