N-CS159: In-Depth Analysis & Comparison of Parallelism Techniques in Python & Go

Hassanin, Ziad Mansour M
Computer Science Department
San Jose State University
San Jose, CA 95192
669-899-7110

ziad.mohamedhassanin@sjsu.edu

ABSTRACT

This paper aims to give more practical experience regards parallelism techniques; we will have Python & Go but the same procedure can be used with any language. First, we will discuss some important concepts and opinions. Such as the popular say among young developers that compiled languages is better than interpreted ones. This is important so to remove any bias and make the best out of any tool. Then We will discuss how concurrency is not the same as parallelism. Secondly, we will start with Python and its parallelism tools e.g., Threads and Subprocesses. Also, we are going to discuss some python internals as these concepts will help us better understand and predict the behavior of the language. Such as the popular Python Global Interpreter Lock – GIL. Finally, we will experiment with Go the language the advertise itself with its concurrency capabilities. We will start explaining what is a routine, then how to spawn multiple goroutines. After that we will move to the Go Scheduler and take a deep dive into its behavior over the different Go versions especially go1.18 and go versions prior to go1.15. We will end with Go channels and how multiple channels are managed in Go code, also some concurrency patterns e.g., Worker Pool Pattern.

1. INTRODUCTION

The lack of practical experience, especially among young developers. Can definitely limit how much they understand the theoretical concepts they studied. For example, it is quite popular among young developers that interpreted languages are slow and inefficient. They are correct in case of comparing Python and Cpp performance with different sorting algorithms. Ignoring the speed and cost of development, the technical dept introduced with writing code in a compiled less readable language like Cpp compared to Python.

For example, the backend codebase of Instagram is entirely built in Python – Django. “If we rewrote the entire codebase of Instagram in a compiled language like Cpp we will get about only 6% increase in performance, without mentioning all the cons of such a decision like the high cost of Cpp development for example or the learning curve of the language. While we were able to achieve 45% increase in performance shifting our attention to a more scalable infrastructure.” [Lisa Guo, Scaling Instagram]

Another concept, that even some senior developers seem to struggle with is that concurrency is not the same as parallelism. Concurrency is about dealing with lots of things at once. While Parallelism is about doing lots of things at once.

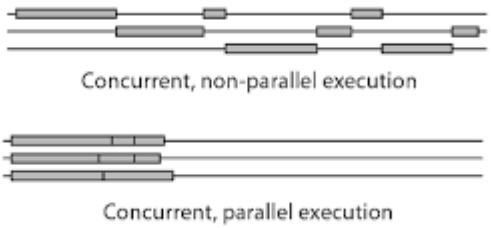

Figure 1. Concurrency in Single core, Parallelism on multicore. [Apurva Thorat, Concurrency & Parallelism]

2. PYTHON INTERNALS

2.1 First Look

We will start with a simple counter program that starts from 0 and counts until 50 million. The code can be found in the GitHub repository ZiadMansourM/ParrallelPythonGo.

```
---------------> Squential
Function 'sequential_exec' executed in 2.2140436172485355
---------------> Threads
Function 'parallel_exec' executed in 4.355177640914917s
---------------> SubProsses
Function 'parallel_exec' executed in 1.2264466285705566s
```

Figure 2. Sequential & Parallel execution [H Ziad, ZiadMansourM/ParrallelPythonGo/gil.py]

For Figure 2 we can see that implementation that used the threads had the worst time then the sequential execution came second and finally the parallel execution that used the subprocess came with the fastest execution time. We will understand this behavior in the next subsections.

2.2 Python Objects

Everything in Python is an Object, mainly consists of two important things a reference counter and object type variable. What is important for us is the reference counter variable.

```
100  struct _object {
101      _PyObject_HEAD_EXTRA
102      Py_ssize_t ob_refcnt;
103      PyTypeObject *ob_type;
104  };
```

Figure 3. Base Python Object [python, python/cpython/Include/object.h]

This reference counter needs to be protected from race conditions. Otherwise, memory leak could happen in case if the counter were bigger than its actual value. This will prevent the counter from reaching zero so it will never be released. As the Python runtime releases objects when its reference counter reaches zero.

What is even worse it the reference counter was less than its actual value. This will cause the object to be released while some other object is still referencing it somewhere. You would be really lucky if your program crashes. Or you will end up with unpredictable weird behavior, that is extremely hard to reproduce or even notice at some times.

2.3 Python GIL

For the above reasons listed in Section 2.2 the Python Global Interpreter Lock – GIL was introduced in 1992. It is in simple words, a mutex or lock that allows only one thread to take control over the Python interpreter.

The other suggested solution was to add locks for the reference counter of each Python object. Which will in return cause so mush deadlocks all around the code. This is why the GIL was the best solution on hand.

This lock will prevent running multiple threads in parallel and would only allow them to run concurrently. Many people heated the GIL for this behavior. And asked to be removed from Cpython. They also tried to implement some python interpreters that don't have the GIL nor the reference counter instead they will have normal garbage collection like Jython written in Java, IronPython written in .Net and PyPy written with the Jit compiler.

"The design decision of the GIL is one of the things that made Python as popular as it is today. Actually, Indeed part of the reasons Python is successful today because it had the GIL" [Larry Hastings, Python's Infamous GIL]. This demonstrate that the GIL is not a bug, but developers have been abusing it so far.

2.4 Python Threads

Python threads don't correspond to hardware threads, at least in the Cpython implementation. They are also managed by the Python runtime, not the OS schedular. The runtime switches between the threads every 100-byte code executed. They are more suitable in I/O bounded tasks. In Figure 3, the program was CPU intensive this is why threads had the worst execution time of the three methods. As, it is the same as the sequential but adding the context switching time between the threads each 100-byte of code.

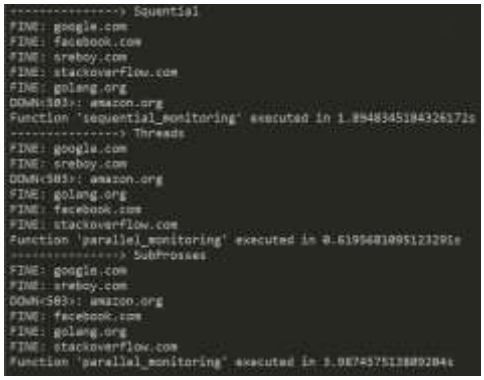

Figure 4. I/O bound [H Ziad, ZiadMansourM/ParrallelPythonGo/thread_subprocess.py]

In Figure 4, the program was I/O bound. This is where python threads shine actually. While waiting for a response it is better to spawn some other requests. This is quite the opposite of what is happening in the sequential implementation. Where the whole code is blocked waiting for a response from the network call. Consequently, wasting resources.

2.5 Python Subprocesses

In Figure 2, we noticed that the subprocesses implementation had the fastest execution time. And in Figure 4 we noticed that it had the worst execution time.

Python Subprocesses are implemented from the multiprocessing pkg from a class called Pool. This is pool of Subprocesses, basically it will spawn another python processes each has its own processes consequently its own interpreter and GIL. But, spawning processes is a bit slower than generating threads. And, creating pools has a certain amount of overhead.

This explains why in Figure 2, it had the fastest execution time. AS, it makes a good use of 4 cores, which are the same number of the chunks the iterable passed was divided into see the code for more details. Also, as it is CPU bound this is exactly where subprocesses fit. Without forgetting the most important part, which is that the problem by its nature is able to be divided into chunks and run in parallel and still get the same results.

In Figure 4, because of the overhead of creating the pool and that each process was waiting for the I/O call that it made, wasting four cores. Also, that generating processes is mush more slower this all resulted in the execution time we saw in Figure 4. Which was even worse than the sequential implementation.

2.6 Multiprocessing API

The multiprocessing package has a class called Pool which is a pool for subprocesses as we stated previously in section 2.5. the same package has a class called ThreadPool which subclasses Pool. This enabled us to have the same API for threads and subprocesses. I made a good use or this in the three scripts found on our repo. The GIL.py, thread_subprocess.py and the map.py script.

The API contains four methods: map, map_async, imap, imap_unordered. They can be divided onto two groups "map, map_async" and "imap, imap_unordered" the main deference between the two groups are: 1- The way they consume the iterable or the data you pass to them. 2- the way and when they return or yield the results back to you.

map, assuming your iterable is not a list already, it will start by converting your iterable into a list. Then it will start breaking it into chunks. After that it will start sending those chunks into workers processes in the pool. This is a huge performance increase than looping over the iterable itself and passing each item between processes one at a time. Especially, if the iterable is large. But, converting the iterable into a list have a very high memory cost. As the list will be stored in memory. You won't be able to see the results until all processes are finished

imap, this method will use the iterable as is. And, will not convert it into a list. Even it will not cut it into chunks. Unless, you specified otherwise by passing the chuncksize argument to imap. This means lower performance but less memory usage. It will yield the results as soon as its finished. Notice: imap keeps order.

imap_unordered, is the same as imap but it will return the results in a random order according to imap_unordered docs. You can think of it as it will return whatever finishes first.

map, is internally implemented as map_async(…).get(). Your code will continue executing after the async_map call and will not block on the map method call as in map method. But you won't be able to retrieve any of the results until all of them finish.

Figure 5. three processes sleep for either [1, 5, 3] seconds [H Ziad, ZiadMansourM/ParrallelPythonGo/map.py]

From Figure 5, we can recommend using imap, imap_unordered. In the following cases: 1- you want to start processing as soon as the results are done. 2- your iterable is large to the point where converting it into a list will cause to use insane amount of the memory, threating your program to crash.

3. GO INTERNALS

3.1 First Look

If you want to master the feature of any languages, starting by its internals is a good start. You won't be able to write reliable Go parallel code if you are not confident enough about what is going on under the hood. A quick introduction for Go. Go is a compiled language that can be compiled into one binary file which make it pretty easy to deploy. Which is a good feature also for scalability and parallelism. Also, Go is known for its code readability.

When you run a main.go file, you actually create a go routine called the main routine. This is run and terminated by the Go runtime.

Figure 6. Sequential Goroutine [H Ziad, ZiadMansourM/ParrallelPythonGo/01main.go]

3.2 Go Routines

You can run multiple go routines by using the keyword go in front of a function call. Goroutines don't correspond to OS threads. As OS threads starting size can be from 1000 till 2000 Kbyte. While a goroutine can have a starting size from 2 KB and can grow in size. This enables us to run thousands and even millions goroutines.

The Goroutines are controlled by the Go schedular, not the OS scheduler. It has some especial behavior we will discuss in details. In the next section.

3.3 Go Schedular

The Go scheduler will spin the first go routine inside the main routine and will keep it running until one of two conditions happens: 1- The Goroutine finishes executing. 2- The routine gets blocked waiting on I/O whether writing to a file or waiting for a network response.

For Go version less than go1.15 the go runtime will run the routines concurrently by default on only one core as the GOMAXPROCS environment variable was set to one as in the following: runtime.GOMAXPROCS(1), this means that the go runtime is allowed to use only one logical core. But, starting from go version later than go1.15 the GOMAXPROCS environment variable is set to the number of logical cores on your machine.

As the Go scheduler gets better and smarter, they will remove modifying GOMAXPROCS environment variable.

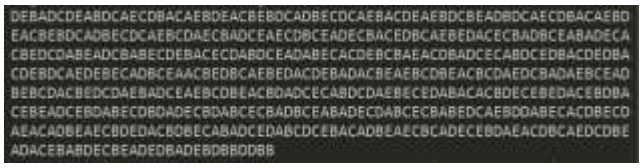

Figure 7. Concurrent Goroutine with blocking call [H Ziad, ZiadMansourM/ParrallelPythonGo/02main.go]

Figure 7, shows a multiple goroutines running concurrently on a single core. Each go routing of them makes a blocking call using time.Sleeep(time.Second). This explains why they are alternating while running.

Figure 8. Concurrent Goroutine without blocking call [H Ziad, ZiadMansourM/ParrallelPythonGo/03main.go]

Figure 8, is the same code as in Figure 7. But I removed the blocking call. This confirms that any go routine will keep running until one of two conditions happens: 1- The Goroutine finishes executing. 2- The routine gets blocked waiting on I/O whether writing to a file or waiting for a network response.

3.4 Sync Package

There is a package call "Sync". This package come in handy when trying to synchronize goroutines. It has three methods Add, Done, Wait.

As soon as, the main routine finishes all the child Goroutines will get killed, to prevent this from happening the main routine has to wait until all the child routines finish execution. Using the Sync package, we will register each goroutine upon creation with the waitgroup.Add(1). This will increase the counter of the wait group

one. After the Goroutine finishes execution. It should call waitgroup.Done(). This will decrement the counter of the wait group by one. Finally, before the main routine ends you must use the waitgroup.Wait(). It will block the main routine until the wait group counter reaches zero.

This can also be used to synchronizes go routines any place in the codebase not specifically in the main routine.

3.5 Channels

In Go there is a popular say that “Don't communicate by sharing memory; share memory by communicating.”. That means Don't overengineer Inter-Thread Communication by shared memory, complicated, error-prone synchronization primitives but instead use message-passing between goroutines (green threads).

Also, Channels can be defined as simple queue. It is a simple message passing pattern. It can also have some buffer locations inside it. The go compiler and linter also provides some static analysis while writing code to validate that you will not send on a read-only channel nor receive on a send-only channel.

Whether you are trying to share a simple int or a complex data structure. The communication in Go states that you modify then send through the channel. Giving away ownership by sending the value or a pointer to a different goroutine via the channel. But take care of passing pointers not recommended and can cause insane amount for hardly to debug bugs. In short there is no shared space, each goroutine only sees the portion of memory it owns.

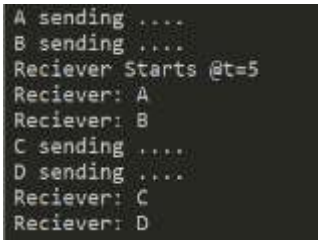

Figure 9. Channel with buffer equal one [H Ziad, ZiadMansourM/ParrallelPythonGo/05main.go]

In Figure 9, the channel capacity is one. So only A will be stored. Then B will be blocked waiting as it has no place inside the channel nor there is no one on the other side consuming what is inside the channel. Until t equals five, The receiver wakes up and start consuming from the channel. Allowing the other Goroutines to be able to send in the channel.

Take care if you forgot to close a channel this will cause it to deadlock waiting to receive something. Always remember to close from the sender side as well.

3.6 Select Channels

Managing multiple channels can be a pain in the nick. But, luckily Go got us with a useful tool called Select. The need for the Select tool arises from that the receive call is always blocking so if you have two channels to receive from if you blocked on one of them you want be able to receive from the other channel as your are blocking on the first one, this will slow your program by multiple’s of the rate of send on the first and second channel combined adding also all the positive feedback effect. As the sender also blocked because you aren’t receiving. This if you didn’t cause deadlock because in this case there is a high probability of deadlocking.

Figure 10. Receive from mutable channels with Select [H Ziad, ZiadMansourM/ParrallelPythonGo/06main.go]

In Figure 10, the code will be able to receive from both the channels without blocking waiting on any of them.

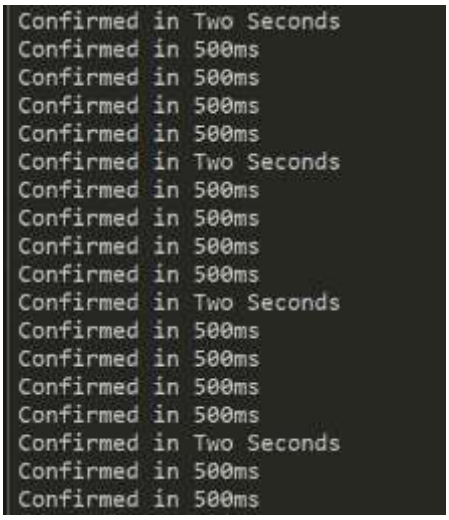

Figure 11. Receive from mutable channels with Select [H Ziad, ZiadMansourM/ParrallelPythonGo/06main.go]

In Figure 11, the runtime was able to receive from the both channels equally at the same time.

3.7 Worker Pool Pattern

There are so many concurrency patterns, But one of the most famous ones are the Worker Pool Pattern. We will have two channels jobs and results our program is trying to calculate the Fibonacci numbers until 40. A worker reads from jobs channel and then calculate is and added it to the results channel.

Having more workers would mean faster execution on condition that the code runs on multicores. If your go version larger than go1.15 you shouldn’t worry about it the schedular after go1.15 is smart enough to do it if the program really needs it. As Toy programs running on a toy machine get toy results. And the scheduler won’t fall for it and waste resource by running them in multicore environment.

So in this code we will try to push the code to its limits. And to limit the factors that can affect the speed of execution to only the number of workers, It is a tough thing to do as we have to close the channels probably and especially the results channel as closing

the results channel. Means waiting for all the workers to finish by synchronizing them with the sync package then close it. This is a bottle nick and will slow the execution really hard because of waiting but we can over come this by some simple tricks like hardcoding the number of items that is going to be received now we don't have to close the channel and no place for deadlock while receiving.

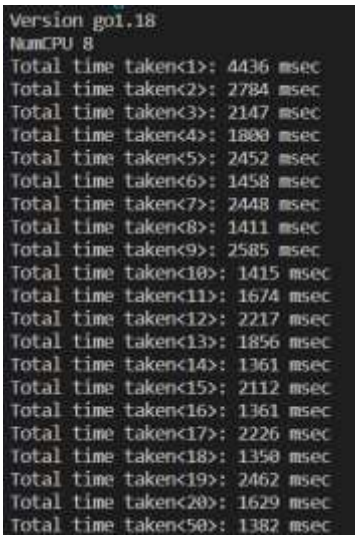

Figure 12. Measuring execution time using various num of workers [H Ziad, ZiadMansourM/ParrallelPythonGo/08main.go]

In Figure 12, we tested various number of workers from one till twenty and then fifty workers. And calculated the execution time for each.

4. SUMMARY/CONCLUSION

Don't underestimate the tools by a shallow look at them. Start by a tool that you are comfortable using it and then dig deeper in its internals. Find you the available built in tools and the limitations on the language capabilities, and most importantly be patient while searching for a proper solution for these limitations. Don't just jump to a new tool. Sometimes opening the codebase of your tool is super helpful if it is Open Source, so train yourself to do it.

5. REFERENCES

[1] Adept, G. [Grow Adept]. (2022, March 29). Golang Beyond the Basics [Video]. Golang Beyond the Basics. https://www.youtube.com/playlist?list=PLDZ_9qD1hkzNtVRH6oLhmqQqz2Cb9O8e3

[2] Guo, L. [InfoQ]. (2017, July 14). Scaling Instagram Infrastructure [Video]. QCon London 2017. https://www.youtube.com/watch?v=hnpzNAPiC0E&t=1975s

[3] Hastings, L. [PyCon Ireland]. (2015, October 25). Python's Infamous GIL [Video]. PyCon Ireland 2015. https://www.youtube.com/watch?v=KVKufdTphKs&t=737s

[4] Kennedy, W. (2014, January 29). Concurrency, Goroutines and GOMAXPROCS. Ardanlabs. Retrieved April 27, 2022, from https://www.ardanlabs.com/blog/2014/01/concurrency-goroutines-and-gomaxprocs.html

[5] McGranaghan, M. (2022, January 1). *Go by Example*. Gobyexample. Retrieved April 27, 2022, from https://gobyexample.com/