CS263 Final Project: Python Multithreading and GIL

Karl Wang

changhai\_wang@ucsb.edu

# Introduction (objective & contribution)

The Python standard library provides concurrency through the multithreading library. However, CPython, the standard Python implementation, only executes a single thread at a time. In this project I investigated the practical effects single threaded execution on concurrent programs, compared CPython's performance on concurrent programs against other Python implementations, and dug into CPython's implementation of Global Interpreter Lock (GIL), the mechanism used to facilitate this behavior.

# Background

A recent trend in CPU development is the accelerated growth on the number of cores. Top end workstation CPUs have reached as much as 64 cores or 128 threads. The existence of GIL poses issues for CPU-bound Python programs as they are locked away from most of these computing power. It is thus worth examining the effects of GIL and it’s implications.

Before analyzing GIL’s effect, it is worth reviewing why it’s needed in the first place. The CPython documentation lists several hurdles that has prevented replacements for GIL: The alternative solution, if it exists, must be fast, simple, and compatible with current features. In particular, the alternative GC must not compromise the performance on single-threaded workloads. PyPy’s FAQ have also noted that while GC is less of a problem for them, they would still need to add fine-grained lock through the whole interpreter.

# Experiments (implementation)

To measure the performance penalties under GIL, I constructed several synthetic tests that represent the different types of concurrent workloads and ran them using different Python implementations.

For the tasks, I made CPU intensive workloads which does repeated arithmetic operations, Disk intensive workloads which does repeated disk writes, and network intensive workloads which grabs pages over HTTP. The CPU-bound tasks represent the worst-case scenario for CPython, as it benefits from true multithreading the most. The network-bound task represents the best case for CPython, as GIL is released while waiting for Network IO.

For the implementations, I chose CPython 2.7/3.6, the standard implementation for Python that uses an interpreter; PyPy 2.7/3.6, which uses JIT compiler; Jython (python 2.7 only), which runs in JVM; and IronPython (python 2.7 only), which uses CLI. Both CPython and PyPy uses GIL, where as Jython and IronPython does not.

Another thing worth noting is that Jython and IronPython have significant compatibility issues that reduced the number of tasks that could be used. IronPython could not do disk read properly, and neither Jython nor IronPython can flush writes to disk. As a result, I only include disk write. Lastly, Neither Jython nor IronPython has the multiprocessing module, as it was a designed as a drop-in replacement for the multithreading module that allows for parallel execution. Since threads can execute on parallel on Jython and IronPython, they have not included this module.

The results were gathered on a machine with Ryzen 5 3600x, which as 6 cores and 12 threads, 16GB DDR4 3200, and an SSD. All Python implementations were run in Windows Subsystem for Linux (WSL) except IronPython, which depends on the .NET Framework and was run in the host Windows system. WSL may have contributed to the noise in the disk data, as it is known that WSL incurs additional latency when writing to disk.

# Results (findings)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| # threads | CPython | CPython3 | PyPy | PyPy3 | Jython | IronPython |
| 1 | 1.78 | 2.75 | 0.20 | 0.20 | 1.71 | 2.32 |
| 2 | 3.69 | 2.84 | 0.22 | 0.23 | 0.88 | 1.54 |
| 4 | 5.33 | 2.88 | 0.24 | 0.27 | 0.46 | 1.01 |
| 8 | 6.33 | 2.91 | 0.28 | 0.36 | 0.37 | 0.83 |
| 16 | 6.17 | 2.90 | 0.41 | 0.58 | 0.34 | 0.98 |
| 32 | 5.80 | 2.90 | 0.55 | 0.82 | 0.31 | 0.97 |

Table : CPU Bound Task

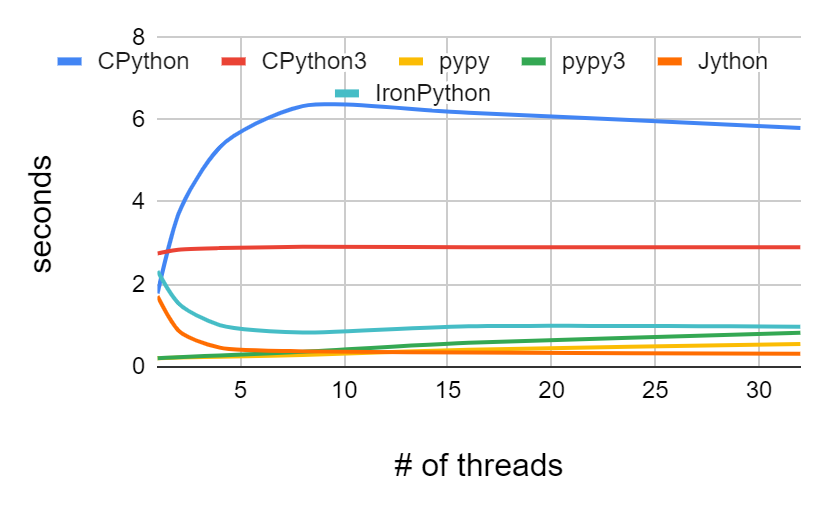


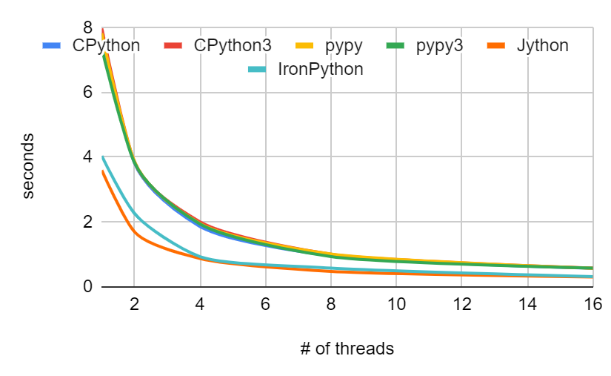
Figure : CPU Bound Task

For the CPU intensive task, the results are quite unexpected. The CPython 3 results are the most normal. Since CPython executes one thread at a time, the one would expect the execution time to stay the same or to slightly increase due to the overhead of starting new threads. This is exactly how CPython3 behaves. Jython and IronPython also behaved sensibly as well. Both implementations started slow and sped up as thread count increases. I was not expecting PyPy and PyPy3 to perform quite so well at low thread count. They are almost 10 times faster than CPython for this task. This makes sense however, as this is a synthetic test with one hot loop that executes a few instruction multiple times, which is the exact situation a JIT compiler would shine.

CPython2 performed unexpectedly bad. With single thread, it ran faster than CPython3. However, it quickly exceeded the runtime of CPython3 as the number of threads increased. It ran almost twice as slow as CPython3 at 16 threads.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| # threads | CPython | CPython3 | PyPy | PyPy3 | Jython | IronPython |
| 1 | 7.8281 | 7.9777 | 7.827 | 7.3665 | 3.6 | 4.0352 |
| 2 | 3.8227 | 3.8559 | 3.8692 | 3.8295 | 1.693 | 2.2617 |
| 4 | 1.8599 | 2.0018 | 1.951 | 1.936 | 0.87 | 0.9293 |
| 8 | 0.9498 | 1.0043 | 1.0101 | 0.9294 | 0.473 | 0.5748 |
| 16 | 0.5678 | 0.5771 | 0.5708 | 0.5741 | 0.302 | 0.3119 |

Table



Figure

For the Network IO task, the results are more normal. All implementations benefitted from the additional threads. This is expected, since when CPython 2/3 waits on IO, it voluntarily releases GIL. CPython thus achieves similar performance as multithreaded implementations at high thread count. However, IronPython and Jython are about twice as fast at low thread counts.

It is unclear why there is such a distinct separation between these two group of implementations at low thread counts. In theory, they should perform the same. Part of this may be attributed to the difference between implementations. [[1]](#footnote-1)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| # threads | CPython | CPython3 | PyPy | PyPy3 | Jython | IronPython |
| 1 | 0.002393 | 0.002818 | 0.005522 | 0.005302 | 0.033 | 0.029984 |
| 2 | 0.01036 | 0.002952 | 0.004624 | 0.005308 | 0.011 | 0.011993 |
| 4 | 0.015421 | 0.003345 | 0.010133 | 0.008916 | 0.015 | 0.004005 |
| 8 | 0.024428 | 0.003598 | 0.014252 | 0.012586 | 0.012 | 0.008995 |
| 16 | 0.026647 | 0.004126 | 0.033111 | 0.026707 | 0.012 | 0.022003 |

Table : 10,000 writes

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| # threads | CPython | CPython3 | PyPy[[2]](#footnote-2) | PyPy3 | Jython | IronPython |
| 1 | 0.211126 | 0.255687 |  | 0.046453 | 0.303 | 0.314651 |
| 2 | 1.1622 | 0.284976 |  | 0.321538 | 0.433 | 0.355194 |
| 4 | 1.640436 | 0.291433 |  | 0.711544 | 0.26 | 0.422997 |
| 8 | 2.442421 | 0.297597 |  | 0.349908 | 0.257 | 0.499992 |
| 16 | 2.403734 | 0.30465 |  | 0.052969 | 0.303 | 0.617996 |

Table : 1,000,000 writes

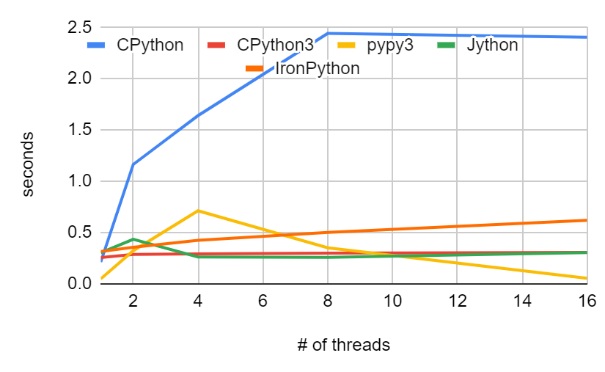
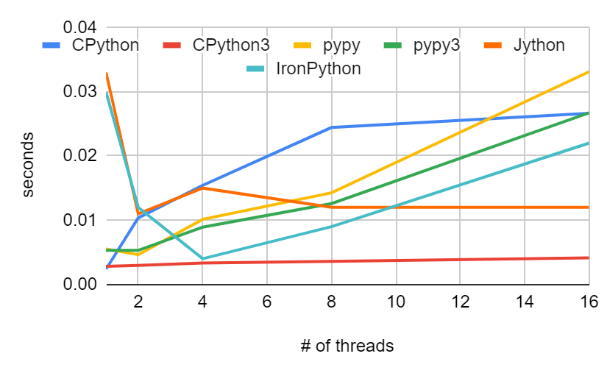


Figure : 10,000 writes Figure : 1,000,000 writes

The disk write task was run with both 10,000 writes (Figure 3) and 1,000,000 writes (Figure 4). The performance characteristic of this task is quite interesting. It seems to have mixed characteristics from both tasks above. This is somewhat expected, since GIL are unlocked for IO operations, even if it’s relatively brief. In this task, CPython3 performed similarly to true multithreaded implementations. However, CPython2 suffered the same performance loss as in the CPU bound tasks.

# Analysis on CPython 2

A consistent theme seems to be that CPython2 performs worse with more threads. Reading about the Python GIL has let me to some information about a switchover of GIL that happened on CPython 3.2/3.3. CPython will always release GIL voluntarily while waiting for IO. However, prior to the switchover, CPython will also release the GIL approximately every 100 Python instructions, causing a lot of thread switching on multicore CPUs. This not only includes switching to other Python threads, but also to other OS threads, since CPython uses real threads and leaves the scheduling to the OS scheduler.

After the switchover, threads will supposedly ask each other to release the GIL via a timeout. TThis allowed the threads to run for long periods at a time, which is great for performance.

I intended to verify this, but I had issues modifying the CPython interpreter. However, the results above seems consistent with this claim, especially seeing how CPython2’s performance stopped degrading at >12 threads.

# Multithreading vs Multiprocessing

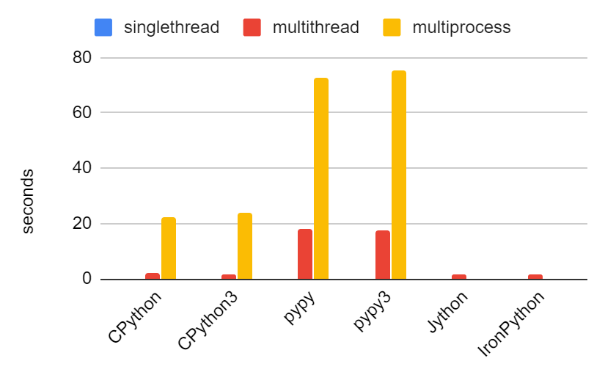


Figure : Quicksort 100 elements with different implementations and concurrency techniques.

The multiprocessing module in Python is designed as a drop-in replacement for multithreading. It provides parallel processing capabilities, but it is also much slower. On CPython 2 & 3, creating a process is about 10 times slower than creating a thread. This could still be viable for long running CPU-bound tasks, especially if is used with a process pool. One major issue however is that the objects are not shared between processes. This will inevitably increase the complexity of programs using multiprocessing.

# Conclusion

Threading is good for IO bound tasks. However, CPython and PyPy’s GIL remain a significant roadblock for CPU bound programs. Some improvements have been made to CPython’s GIL, but one should still look for alternatives when running a CPU bound task.

1. I manually timed the result for Jython as a sanity check. At 1 thread, Jython reported 3.6 seconds, but I measured 6.0 seconds. In comparison, CPython reported 6.6 seconds, and I measured 6.9 seconds. One possible explanation for the similarity in overall execution time is Jython needs to take some time to compile to bytecode. Still, that is a significant overhead, especially for small programs. [↑](#footnote-ref-1)
2. PyPy2’s results for 1,000,000 writes are not included because it reaches 20-30 seconds at 4 threads and 0.06-80 seconds at 16 threads. Given that it performed much more consistently with lower number of writes, this seems to be a problem in PyPy’s write method, and not an issue in the multithreading module. [↑](#footnote-ref-2)