Python threads: Dive into GIL!

PyCon 2011
Pune Sept 16-18
Vishal Kanaujia and Chetan Giridhar
Summary

• Benefit of multi-threaded application grows with ubiquity of multi-core architecture that potentially can simultaneously run multiple threads of execution.

• Python supports multi-threaded applications and developers are flocking to realize the assured gain of multiple cores with threaded applications.

• Unfortunately, Python has significant bottleneck for multi-threading.
Summary...

• Any thread in CPython interpreter requires a special lock (GIL) which results in serial, rather than parallel execution of multi-threaded applications, irrespective of cores availability and design techniques.

• This talk focuses on the problem, dissects the root cause and its implications.
A jaw dropping example!

- A simple python program – single function performing two operations for 10000000 iterations:
  - Divides 2 random numbers from specified range
  - Multiplies 2 random numbers from specified range
  - Called as two different threads on:
    - Single Core
    - Dual Core

<table>
<thead>
<tr>
<th>Python v2.7</th>
<th>Execution Time</th>
<th>User Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Core</td>
<td>55 s</td>
<td>1.108 s</td>
</tr>
<tr>
<td>Dual Core</td>
<td>67 s</td>
<td>3.071 s</td>
</tr>
</tbody>
</table>

22% dip in Execution Time

Increased User Time by 2 secs.
Threads: Fundamentals

• Fundamental to a multi-tasking application
• Smallest possible, independent unit of execution
• Light weight processes (resource sharing including address space)
• Concurrent execution
  ▪ Uni-core processor: Single thread at a time; Time division multiplexing
  ▪ Multi-core processor: Threads run at the same time
• CPU bound and I/O bound
Python Threads

• Real system threads (POSIX/ Windows threads)
• Python VM has no intelligence of thread management (priorities, pre-emption, and so on)
• Native operative system supervises thread scheduling
• Python interpreter just does the per-thread bookkeeping.
Python threads: internals

• Only one thread can be active in Python interpreter
• Each ‘running’ thread requires exclusive access to data structures in Python interpreter
• Global interpreter lock (GIL) provides this exclusive synchronization
• This lock is necessary mainly because CPython's memory management is not thread-safe.

• Result
  – A thread waits if another thread is holding the GIL, even on a multi-core processor! So, threads run sequentially, instead of parallel!
Python threads

• How do Python manages GIL?
  – Python interpreter *regularly* performs a check

  ![Image of a smiley face with a button that says Ticks >0 and a burst that says check]

• A check is done after ‘n’ ticks.
  – It maps to ‘n’ number of Python VM’s byte-code instructions
  – A global counter; Ticks decrement as a thread executes

• As soon as ticks reach zero:
  – the active thread *releases and reacquires* the GIL
  – Signal handling (only in the main thread)

• Effectively, ticks dictate allowed CPU time-slice available to a thread
• Is independent of host/native OS scheduling
• Can be set with `sys.setcheckinterval(interval)`
Python thread: internals

CPU operation → I/O operation → CPU operation

{ Acquire GIL } → { Release GIL } → { Acquire GIL }

CPU bound thread

{ check } → { check } → { check }
GIL: Details and Bottleneck

• GIL is a conditional variable.
• What goes behind the scene?
  – If GIL is unavailable, a thread goes to sleep and wait.
  – At every ‘check’, a thread release the GIL, and tries to re-acquire
• GIL release is accompanied with a request to host OS to signal all waiting threads
• Regular GIL unlock, thread signaling, wake-up, and GIL relock are an expensive series of operations
• **Threads effectively run in the serial order**
GIL: Battle in multi-cores

• Unlike single core, multiple cores allows the host OS to schedule many threads concurrently

• A thread that had just released the GIL, will send a signal to waiting threads (through host OS) and is ready to acquire the GIL again!

• This is a GIL contention among all threads
GIL: Battle continues...

• There is considerable time lag of
  – Communication
  – Signal-handling
  – Thread wake-up
  – and acquire GIL

• These factors along with cache-hotness of influence new GIL owner which is usually the recent owner!

• In a [CPU,I/O]-bound mixed application, if the previous owner happens to be a CPU-bound thread, I/O bound thread starves!
  – Since I/O bound threads are preferred by OS over CPU-bound thread; Python presents a priority inversion on multi-core systems.
New GIL: Python 3.2

• Tries to avoid GIL battle. How?
• Regular “check” are discontinued and replaced with a time-out.
  • Default time-out= 5ms
  • Configurable through sys.setswitchinterval()
• For every time-out, current GIL holder, is forced to release it, signals the waiting threads and, \textit{waits for a signal from the new owner of GIL}.
  – A thread does not compete for GIL in succession
• A sleeping thread wakes up, acquires the GIL, and signals the last owner.
• New GIL ensures that every thread gets a chance to run (on a multi-core system)
Python v3.2: What’s good?

• More responsive threads
• Less overhead, lower lock contention
• No GIL battle
• All iz well😊
New GIL: All is not well

- **Convoy effect** - observed in an application comprising I/O-bound and CPU-bound threads
- A side-effect of an optimization in Python interpreter
  - Release the GIL before executing an I/O service (read, write, send, recv calls)
- When an I/O thread releases the GIL, another ‘runnable’ CPU bound thread can acquire it (remember we are on multiple cores).
- It leaves the I/O thread waiting for another time-out (5ms)!
- Once CPU thread releases GIL, I/O thread acquires and release it again
- This cycle goes on => performance suffers 😞
Convoy effect

- Adversely impacts an I/O thread, if application has a CPU thread(s)
- Voluntary relinquish of GIL proves fatal for I/O thread’s performance
- We performed following tests with Python3.2:
  - CPU thread spends less than few seconds (<10s)!

<table>
<thead>
<tr>
<th>I/O thread with CPU thread</th>
<th>I/O thread without CPU thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>97 seconds</td>
<td>23 seconds</td>
</tr>
</tbody>
</table>
Convoy effect: Python v2?

• Convoy effect holds true for Python v2 also
• The smaller interval of ‘check’ saves the day!
  – I/O threads don’t have to wait for a longer time (5 m) for CPU threads to finish
  – Should choose the setswitchinterval() wisely
• The effect is not so visible in Python v2.0
Comparing: Python 2.7 & Python 3.2

<table>
<thead>
<tr>
<th>Python v2.7</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Core</td>
<td>74s</td>
</tr>
<tr>
<td>Dual Core</td>
<td>116 s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Python v3.2</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Core</td>
<td>55 s</td>
</tr>
<tr>
<td>Dual Core</td>
<td>65 s</td>
</tr>
</tbody>
</table>

On Single Core

- **Execution Time**
  - Python v2.7: 74s
  - Python v3.2: 55 s

On Dual Core

- **Execution Time**
  - Python v2.7: 116 s
  - Python v3.2: 65 s
Solving GIL problems

• Thought #1: reduce the waiting time interval between threads.
  – Caveat: increases the overhead of context switching between threads

• Thought #2: implement GIL with C API extensions
  – Caveat: Lot of rework involved

• Thought #3: allow running of I/O threads with GIL if they are not blocking other threads.
  – Caveat: to be analyzed
Jython: GIL

- Jython is free of GIL
- It can fully exploit multiple cores, as per our experiments
- Experiments with Jython2.5
  - Run with two CPU thread in tandem

<table>
<thead>
<tr>
<th>Jython2.5</th>
<th>Execution time</th>
<th>User time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single core</td>
<td>38 s</td>
<td>0.652 s</td>
</tr>
<tr>
<td>Dual core</td>
<td>32 s</td>
<td>1.524 s</td>
</tr>
</tbody>
</table>

- Experiment shows performance improvement on multi-core system
Conclusion

- Multi-core systems are becoming ubiquitous
- Python application should exploit this abundant power
- Python inherently suffers the GIL limitation
- An intelligent awareness of Python interpreter behavior is helpful in developing multi-threaded applications
- Understand and use 😊
References

• Understanding the Python GIL, http://dabeaz.com/talks.html
• Thread State and the Global Interpreter Lock, http://docs.python.org/c-api/init.html#threads
• Python v3.2.2 documentation, http://docs.python.org/py3k/
Backup slides
Python: GIL

- A thread needs GIL before updating Python objects, calling C/Python API functions
- Concurrency is emulated with regular ‘checks’ to switch threads
- Applicable to only CPU bound thread
- A blocking I/O operation implies relinquishing the GIL
  - ./Python2.7.5/Include/ceval.h
    - `Py_BEGIN_ALLOW_THREADS`
    - `Do some blocking I/O operation ...`
    - `Py_END_ALLOW_THREADS`
- Python file I/O extensively exercise this optimization
GIL: Internals

- The function `Py_Initialize()` creates the GIL
- A thread create request in Python is just a `pthread_create()` call
- `../Python/ceval.c`
- `static PyThread_type_lock interpreter_lock = 0; /* This is the GIL */`
- o) thread_PyThread_start_new_thread: we call it for "each" user defined thread.
- calls `PyEval_InitThreads()` -> `PyThread_acquire_lock()`
GIL: in action

- Each CPU bound thread requires GIL
- ‘ticks count’ determine duration of GIL hold
- new_threadstate() -> tick_counter
- We keep a list of Python threads and each thread-state has its tick_counter value
- As soon as tick decrements to zero, the thread release the GIL.
GIL: Details

def thread_PyThread_start_new_thread() ->
void PyEval_InitThreads(void)
{
    if (interpreter_lock)
        return;
    interpreter_lock = PyThread_allocate_lock();
    PyThread_acquire_lock(interpreter_lock, 1);
    main_thread = PyThread_get_thread_ident();
}