IMPROVING PERFORMANCE OF THREADS IN PYTHON USING C/C++ EXTENSIONS

Author: Olga CASIAN
Scientific advisor: lect. sup. Dumitru CIORBĂ
Technical University of Moldova
Email: dae.eklen@gmail.com, dumitru.ciorba@ati.utm.md

Abstract: *Although Python provides all the synchronization primitives and communication mechanisms one can need for fine-grained management of concurrency, due to the design of the CPython interpreter it is impossible to execute the code on multiple CPUs at a time. This is caused by Global Interpreter Lock (GIL), the mechanism that guaranties that only one thread can execute Python byte code at a time. The article describes the origin of this problem and provides a solution that can significantly improve performance by using C/C++ code extensions, thus benefiting from the speed of compiled languages.*

Index Terms: *Python threads, Global Interpreter Lock, C/C++ extensions, performance.*

1. Introduction

Python has both lower and higher level threading module interfaces which makes possible to execute code in multiple threads of control sharing their global data space. The language provides a big variety of different synchronization primitives and communication mechanisms.

However, the thread performance tests mentioned in [1] proved that executing code in multiple threads does not imply code execution on multiple processors, thus having increase in performance. The tests showed that running code in two threads on dual-core processor is 1.8 times slower than running the same code in sequential manner. Even disabling one of the CPUs cores gives 1.5 times slower result.

2. Global Interpreter Lock

The described behavior of threads is caused by Global Interpreter Lock (GIL), the mechanism used in the CPython interpreter that guaranties that only one thread executes Python byte code at a time [2]. The GIL was introduced to simplify CPython implementation and maintenance by making the object model implicitly concurrent safe.

The implementation of GIL is based on signaling: each running thread acquires the lock and it is released basically only on the blocking I/O operations. The Python itself has no thread scheduler leaving this to the operating system running on the computer. The interpreter performs periodic check operations to deal with the CPU-bound threads. During these checks release and reacquisition of the GIL takes place giving the possibility to other threads to run. While checking periodically the CPython interpreter locks a mutex and signals to condition variable. Because other threads are waiting there is a need in extra pthreads processing and systems calls for delivering the signal that causes performance issues during the test [1].

An attempt to remove the GIL was taken in one of the patches in 1996. Despite the expected result performance of GIL-less Python suffered more than in seven times [3]. The GIL implementation changed through better since Python 3.2 release where multiple threads have almost the same performance as sequential code execution [4], in spite Python 2.7 release is still considered to be the most widely used one.

There are some standard or third-party extension modules in Python that are able to release the GIL while performing computationally-intensive tasks as computing hash functions or solving complex mathematical operations. However standard *thread* and *threading* modules are not such extensions.

3. C/C++ extensions

Writing extension modules in C/C++, thus avoiding GIL, is a solution for previously described problem.

Each programming language has its own strength. One can use Python everywhere flexible, fast and easy to maintain development is needed. Compiled languages as C or C++ are often complex to program;

however they are optimized for the speed of execution. Mixing two different language types gives not only a gain in the execution speed, but also the benefits of interpreted environments as rapid development, interactivity, simplicity of debugging and high level of programming.

There are two distinct integration models [5]:

- ***The extending interface*** – for executing C (or other compiled language) library code from Python programs. The model is usually used for optimization or extending language possibilities purposes.
- ***The embedding interface*** – for running Python code from compiled languages programs. The model is mostly used for providing an additional customization layer.

Manual coding C extensions can turn out fairly involved, that is why it is a common practice to use tools that generate all required integration code automatically. There are several extension building tools as SIP, GRAD and SWIG. The least one is one of the most widely used systems by Python developers.

4. SWIG wrapping

Simplified Wrapper and Interface Generator (SWIG) is an open source system that connects programs written in C and C++ with a large variety of high level scripting and non-scripting programming languages as Perl, Python, PHP, Ruby, C#, Java and others. SWIG is mostly used to parse C/C++ interfaces and generate "glue code" required for the target language to call the C/C++ code [6]. The tool started as a small simple project, but the feature set has grown with the contributions of its users.

One should perform the following steps to create a minimal C/C++ extension for Python code:

- ***Write the code in C/C++.*** Let's consider the simple example of *example.c* file that will contain function returning the cube of an integer:

```
#include <stdio.h>
#include <stdlib.h>
int cube(int n){
  return n * n * n;
}
```

- ***Define interface*** file that will be the input to SWIG (optional in some cases). The file should contain declarations of the things one wants to access. Here is the content of *example.i* file:

```
%module example
extern int cube(int n);
```

- ***Build the module.*** This step is operating system dependent and usually consists of a few commands [4] that produce the file that should be compiled and linked with the rest of the program.
- ***Use the module.*** After importing C/C++ modules behave almost the same as those written in Python:

```
import example
print example.cube(3)   # 9
```

5. Conclusion

Despite significant limitations in realization of threads in Python, there are situations when using threads is appropriate. For example they are useful for I/O bound applications where response time for any I/O activity will still be very quick because of host operating system scheduler usage. Certainly, all CPU-bound processing should be left to C/C++ extensions that will release the GIL and significantly improve performance.

Although many projects successful use SWIG, there are some limitations as the lack of variable linking, representation of pointer model, incomplete handling of C++ classes in Python that is important to consider.

References

1. David Beazley, *An Introduction to Python Concurrency*, http://www.slideshare.net/dabeaz/an-introduction-to-python-concurrency, 2009.
2. *Python Software Foundation*, http://docs.python.org/glossary.html#term-global-interpreter-lock, 2011.
3. David Beazley, *An Inside Look at the GIL Removal Patch of Lore*, http://dabeaz.blogspot.com/2011/08/inside-look-at-gil-removal-patch-of.html, 2011.
4. David Beazley, *Inside the New GIL*, http://www.dabeaz.com/python/NewGIL.pdf, 2010.
5. Mark Lutz, *Programing Python, 4th Edition*, 2010, p. 1483-1502.
6. *SWIG community*, http://www.swig.org/, 2011.