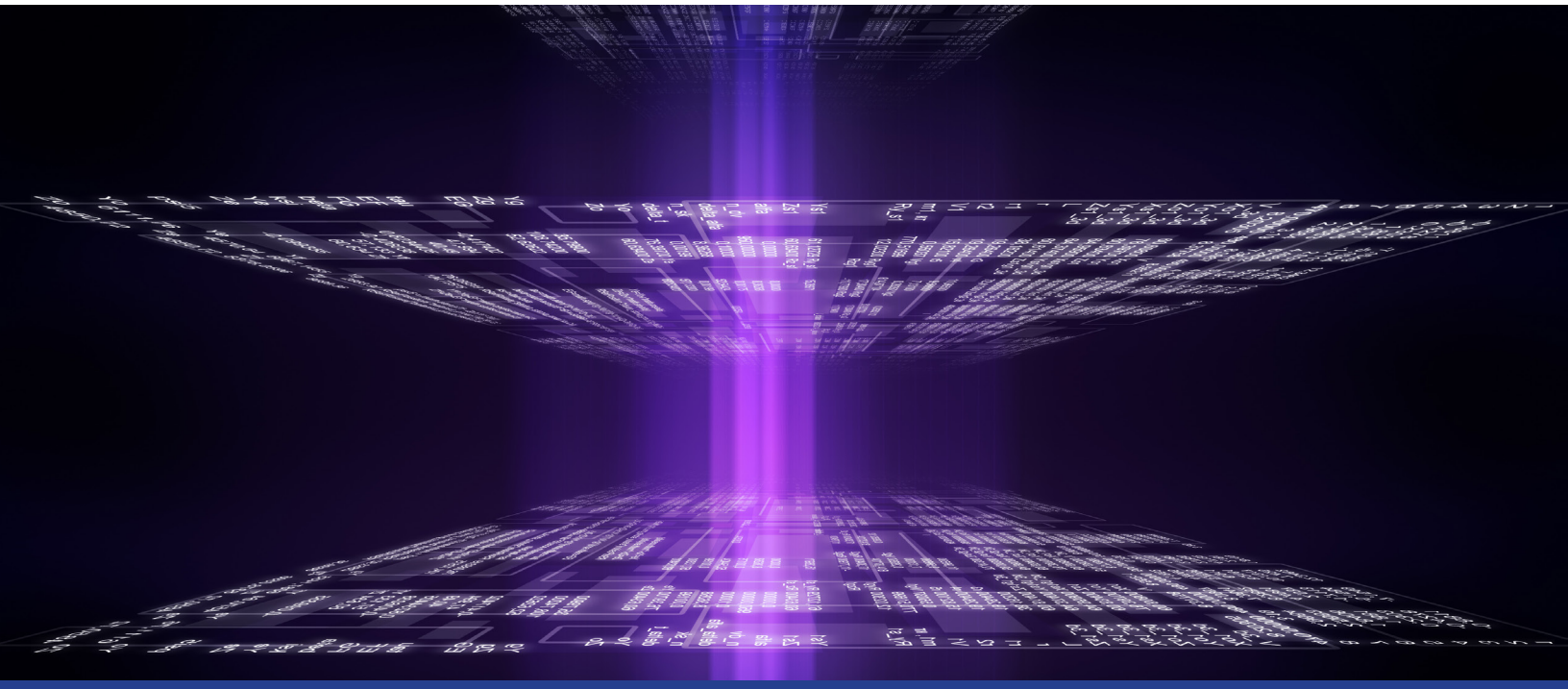

PARALLELISM IN PYTHON*

Dispelling the Myths with Tools to Achieve Parallelism

David Liu, Software Technical Consulting Engineer, and Anton Malakhov, Software Development Engineer, Intel Corporation

Python* as a programming language has enjoyed nearly a decade of usage in both industry and academia. This high-productivity language has been one of the most popular abstractions to scientific computing and machine learning, yet the base Python language remains single-threaded. Just how is productivity in these fields being maintained with a single-threaded language?

The Python language's design, by Guido van Rossum, was meant to trade off type flexibility and predictable, thread-safe behavior against the complexity of having to manage static types and threading primitives. This, in turn, meant having to enforce a global interpreter lock (GIL) to limit execution to a single thread at a time to preserve this design mentality. Over the last decade, many concurrency implementations have been made for Python—but few in the region of parallelism. Does this mean the language isn't performant? Let's explore further.

The base language's fundamental constructs for loops and other asynchronous or concurrent calls all abide by the single-threaded GIL, so even list comprehensions such as `[x*x for x in range(0,10)]` will always be single-threaded. The threading library's existence in the base language is also a bit misleading, since it provides the behavior of a threading implementation but still operates under the GIL. Many of the features of Python's concurrent futures to almost-parallel tasks also operate under the GIL. Why does such an expressive productivity language restrict the language to these rules?

The reason is the level of abstraction the language design adopted. It ships with many tools to wrap C code, from ctypes to cffi. It prefers multiprocessing over multithreading in the base language, as evidenced by the multiprocessing package in the native Python library. These two design ideas are evident in some of the popular packages, like NumPy* and SciPy*, which use C code under the Python API to dispatch to a mathematical runtime library such as **Intel® Math Kernel Library (Intel® MKL)** or OpenBLAS*. The community has adopted the paradigm to dispatch to higher-speed C-based libraries, and has become the preferred method to implement parallelism in Python.

In the combination of these accepted methods and language limitations are options to escape them and apply parallelism in Python through unique parallelism frameworks:

- **Numba*** allows for JIT-based compilation of Python code which can also run LLVM*-based Python-compatible code.
- **Cython*** gives Python-like syntax with compiled modules that can target hardware vectorization as it compiles to a C module.
- **numexpr*** allows for symbolic evaluation to utilize compilers and advanced vectorization.

These methods escape Python's GIL in different ways while preserving the original intent of the language, and all three implement different models of parallelism.

Let's take the general example of one of the most common language constructs on which we'd want to apply parallelism—the for loop. Looking at the loop below, we can see that it provides a basic service, returning all the numbers less than 50 in a list:

```
def test_func(list_of_items):
        final_list = []
        for item in list_of_items:
                if item < 50:
                final_list.append(item)
        return final_list
```

Running this code gives the following result:

```
import random
random_list = [random.randint(0,1000000) for x in range(0,1000000)]
%timeit test_func(random_list)
27.4 ms ± 331 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Python handles the list of items in a single-threaded way under the GIL, since it's written in pure Python. Thus, it handles everything sequentially and doesn't apply any parallelism to the code. Because of the way this code is written, it's a good candidate for the Numba framework. Numba uses a decorator (with the @ symbol) to flag functions for just-in-time (JIT) compilation, which we'll try to apply on this function:

```
%timeit test_func(random_list)
15.7 ms ± 173 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Running this code now gives the following result:

```
%timeit test_func(random_list)
15.7 ms ± 173 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Including this simple decorator nearly doubled performance. This works because the original Python code is written in primitives and datatypes that can be easily compiled and vectorized to a CPU. Python lists are the first place to look. Normally, this data structure is quite heavy with its loose typing and built-in allocator. However, if we look at the datatypes that `random_list` contains, they're all integers. Because of this consistency, the JIT compiler of Numba can vectorize the loop.

If the list contains mixed items (e.g., a list of chars and ints), the compiled code will throw a TypeError because it can't handle the heterogeneous list. Also, if the function contains mixed datatype operations, Numba will fail to produce a high-performance JIT-compiled code and will fall back to Python object code.

The lesson here is that achieving parallelism in Python depends on how the original code is written. Cleanliness of datatypes and the use of vectorizable data structures allow Numba to parallelize code with the insertion of a simple decorator. Being careful about the use of Python dictionaries pays dividends, because historically they don't vectorize well. Generators and comprehensions suffer from the same problem. Refactoring such code to lists, sets, or arrays can facilitate vectorization.

Parallelism is much easier to achieve in numerical and symbolic mathematics. NumPy and SciPy do a great job dispatching the computation outside of Python's GIL to lower-level C code and the Intel MKL runtime. Take, for example, the simple NumPy symbolic expression, `((2*a + 3*b)/b)`, expressed below:

```
import numpy as np
a = np.random.rand(int(1e6))
b = np.random.rand(int(1e6))

%timeit (2*a + 3*b)/b
8.61 ms ± 108 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

This expression makes multiple trips through the single-threaded Python interpreter because of the structure and design of NumPy. Each return from NumPy is dispatched to C and returned back to the Python level. Then, the Python object is sent to each subsequent call to be dispatched to C again. This back-and-forth jumping becomes a bottleneck in the computation, so when you need to compute custom kernels that can't be described in NumPy or SciPy, numexpr is a better option:

```
import numexpr as ne
%timeit ne.evaluate('(2*a + 3*b)/b')
2.22 ms ± 52.7 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

How does numexpr achieve nearly a 4x speedup? The previous code takes the symbolic representation of the computation into numexpr's engine to generate code that works with the vectorization commands from the vector math library in Intel MKL. Thus, the entire computation stays in low-level code before completing and returning the result back to the Python layer. This method also avoids multiple trips through the Python interpreter, cutting down on single-threaded sections while also providing a concise syntax.

By looking at the Python ecosystem and evaluating the different parallelism frameworks, it's evident that there are good options. To master Python parallelism, it's important to understand the tools and their limitations. Python chose the GIL as a design consideration to simplify framework development and give predictable language behavior. But, at the end of the day, the GIL and its single-threaded restrictions are easy to sidestep with the right tools.

Learn More

- **Intel® Distribution for Python**
- **Intel® Math Kernel Library**

BLOG HIGHLIGHTS

10 Huge Benefits of Edge AI & the Software Tools to Deliver Them

CHARLOTTE DRYDEN, INTEL CORPORATION

Artificial Intelligence (AI) continues to show up in our everyday lives, but its presence is gentle and welcome, largely due to the advancements in Edge AI. Many AI use cases are best suited for the edge where processing happens at or close to the data source, lowering costs, reducing application or service latency, improving reliability and increasing data privacy.

Whether we realize it or not, Edge AI technologies—seen and unseen—provide huge benefits in a world that's digitally connected, 24x7. This rapid advancement of Edge AI is not because of one or two "killer apps"—new solutions and usages continue to emerge all the time.

Read more >