Procuring Performance in Python

Ian J. Bertolacci
Email: ian.bertolacci@gmail.com
Twitter: @ianbertolacci
Github: github.com/ian-bertolacci

About this talk

- General overview of *some* modules and tools that you can use to write more performant code.
- Share!
- I am not an expert in fast Python (yet).
- Code available on github
  - github.com/ian-bertolacci/procuring_python_performace_talk

General Classes of Tools

- Compilers/Interpreters
  - CPython
  - PyPy
- Low-level backed APIs
  - NumPy
- Parallel modules
  - Multiprocessing
  - Threading
- Low-level tie-ins
  - Cython * compiler-y
  - PyCUDA

Why is Python slow?

- Because the implementation is slow.
    - Python's semantics are difficult to provide without a runtime that includes lots of overhead.
- Where does this runtime overhead come from?
    - Dynamic typing
        - More memory requirements
        - Indirect accesses
        - explicit checking
    - Non-contiguous list elements
    - Large integers

Example of Overhead

Python code:

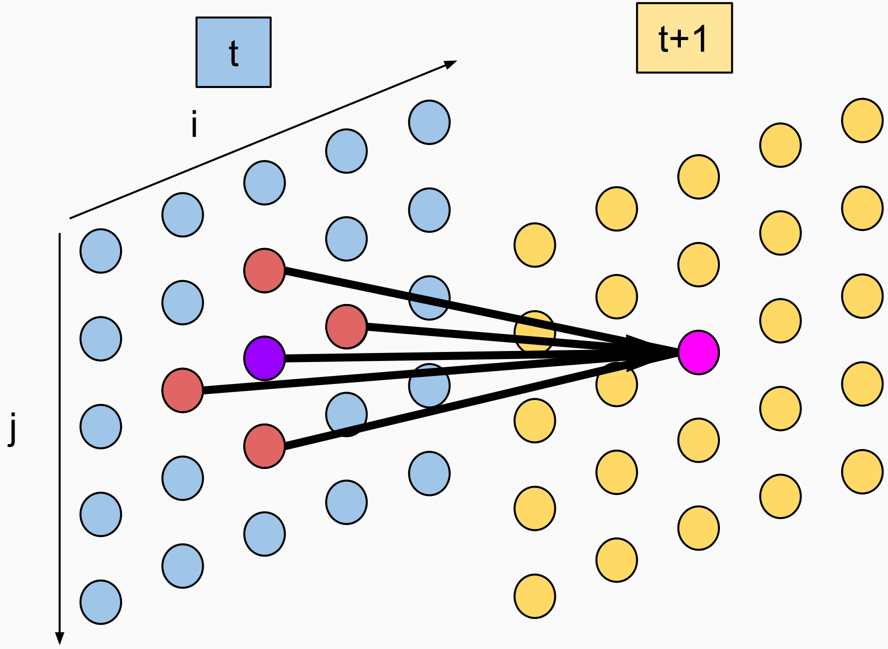
```
a = 1
b = 2
c = a + b
```

This example taken from "Why Python is Slow" by Jake VanderPlas (https://jakevdp.github.io/blog/2014/05/09/why-python-is-slow/)

1. Assign `1` to `a`
   a. Set `a->PyObject_HEAD->typecode` to integer
   b. Set `a->val = 1`
2. Assign 2 to b
   a. Set `b->PyObject_HEAD->typecode` to integer
   b. Set `b->val = 2`
3. call `binary_add(a, b)`
   a. find typecode in `a->PyObject_HEAD`
   b. `a` is an integer; value is `a->val`
   c. find typecode in `b->PyObject_HEAD`
   d. `b` is an integer; value is `b->val`
   e. call `binary_add<int, int>(a->val, b->val)`
   f. result of this is `result`, and is an integer.
4. Create a Python object `c`
   a. set `c->PyObject_HEAD->typecode` to integer
   b. set `c->val` to `result`

Basic Benchmarks

- Two benchmarks are used here:
  - Fibonacci
    - Both recursive and iterative implementations.
    - N = 40
  - Jacobi 2D
    - Stencil computation.
    - 1000^2 grid and 1000 timesteps
      - 5 Giga FLOPS (1000^2 * 5 * 1000)
- Machine:
  - Intel i7-6900K @ 3.20GHz
    - 8 cores / 16 hyperthreads
  - Nvidia GeForce GTX 1080
    - 8.5Gb On package RAM
  - 32 Gb RAM

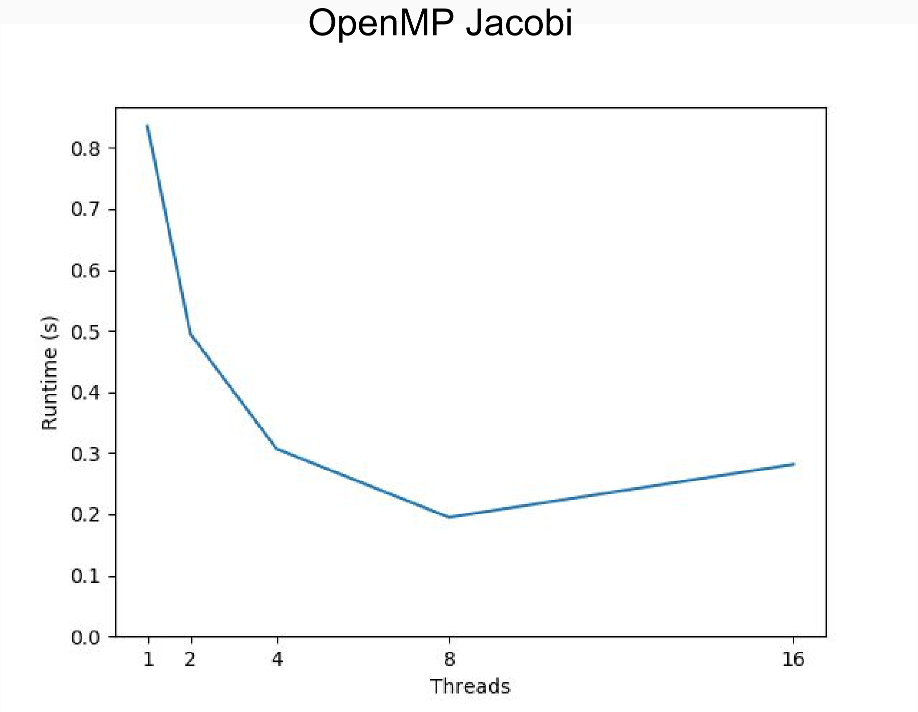


Baseline - Results

- CPython
  - Fibonacci:
    - Iterative:
      - 0.00065 seconds
    - Recursive:
      - 59.2053 seconds
  - Jacobi:
    - 488.244 seconds (~8 minutes)
    - 10.24 Mega FLOPS/second

Baseline - Results

- C
  - Fibonacci
    - Iterative
      - 0.0000003 seconds
    - Recursive
      - 0.433
  - Jacobi
    - Serial
      - 0.897 seconds



PyPy

- Alternative Python interpreter.
  - Probably most popular after CPython
- Written in RPython (a Python derivative). Work your head around that one.
- Pros:
  - Usea a just in time (JIT) compiler that compiles python code to lower code closer to machine level
  - No modifications to code required (except…)
- Cons:
  - Limited support for module using CPython's C-API
    - Support getting better, but performance could vary.
    - May require modified libraries

PyPy - Benchmark Results

- Fibonacci
  - Iterative:
    - 0.00007 seconds
  - Recursive:
    - 5.102 seconds
- Jacobi
  - 9.09s (53.6x faster than CPython Jacobi)
  - 549.67 Mega FLOPS/second

PyPy - Related

- There are a gajillion python interpreters and compilers
  - Jython: JVM
  - Pyston: LLVM
  - Pyjion:
  - Hope
  - Falcon
  - PyDron
  - Nuitka: Compiler, almost all of Python.
  - Shed Skin: Compiler, limited subset of Python.
  - Pythran: Compiler, limited subset of Python
  - GT-Py: Intel's interpreter. Adds OpenMP and OpenACC annotations

NumPy

- Non-standard (but wildly popular) module
- Mainly C backed multi-dimensional arrays and some linear algebra tools
- Pros:
  - Powerful array abstractions.
  - Basis of SciPy (scientific python module filled with magic).
  - Mostly written in C/C++/Fortran (fast).
- Cons:
  - Need to modify code to use it.
    - Not a big deal, but would always like to avoid redevelopment.

NumPy - Benchmark Results

- Jacobi
  - 4.592 seconds (106.324x faster than CPython basline)
  - 1088.850 Mega FLOPS/second

Multiprocessing

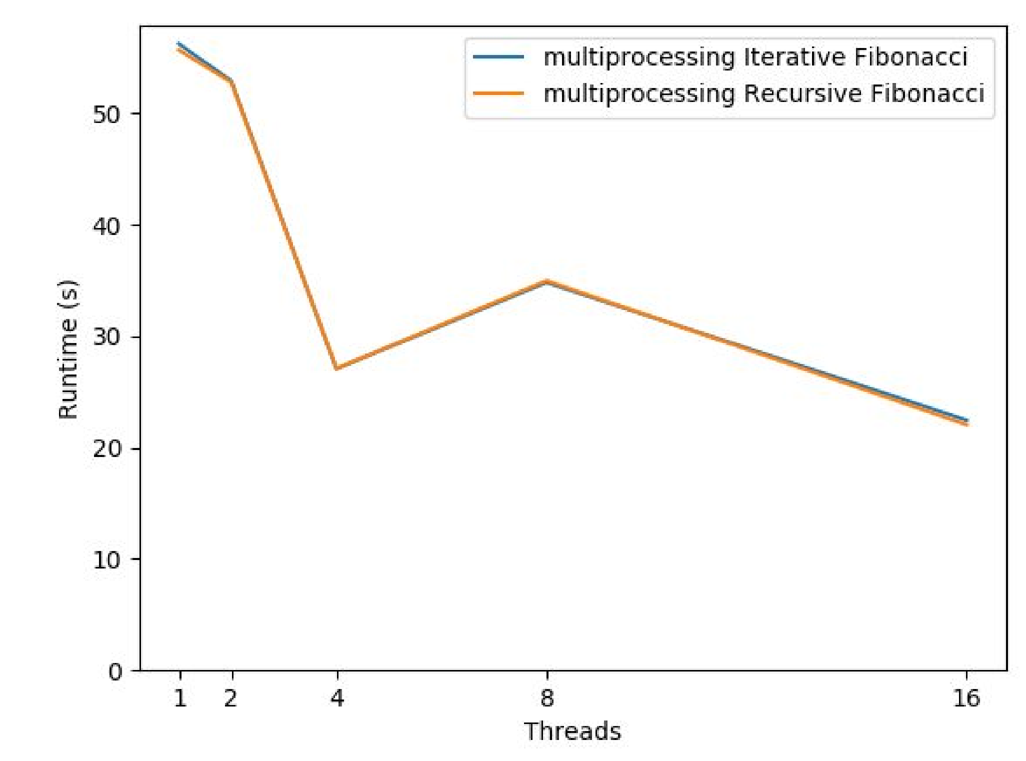
- Standard module.
- Parallelism through processes (not threads).
- Provides mechanisms for usual task-based parallelism.
- Pros:
  - API is fairly standard for a task-based parallelism (create process objects, start, and join them; locks, pipe, semaphore).
  - Provides process pools that can easily map work across them.
  - Supposedly can use multiprocessing on a cluster.
  - Not limited by a Global Interpreter Lock (GIL).
- Cons:
  - Very heavy weight (creates entirely new python interpreter process).
  - Requires you to be quite hands on.

Multiprocessing - Terse Example

```
processes = [
  Process(
    target=work_function,
    args=(work_unit, result_queue)
  )
  for work_unit in work_list
]
# Start all processes
for process in processes:
  process.start()
# Wait for all processes to stop
for process in processes:
  process.join()
```

```
pool = Pool( cpu_count() )
results = pool.map(
  work_function,
  work_list
)
```

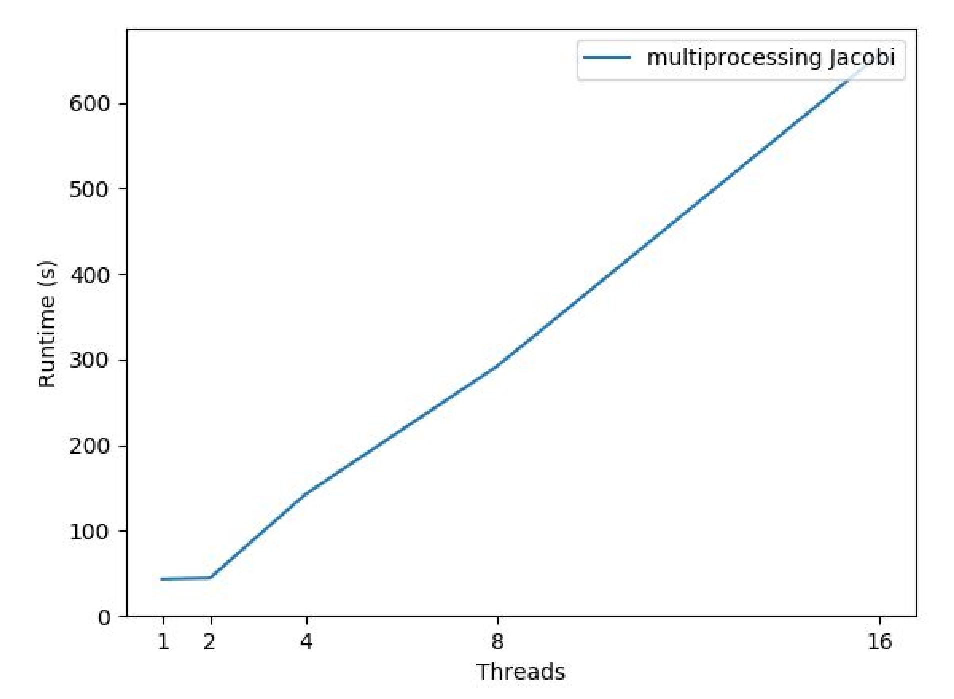
Multiprocessing - Fibonacci Results



Baseline was:

- Iterative: 0.00065 seconds
  - Multiprocessing slower
- Recursive: 59.20 seconds
  - Multiprocessing faster

Multiprocessing - Jacobi Results



Uses NumPy arrays to simplify comms
Baseline was:

- 488.244 seconds
  - Multiprocessing does worse (and keeps getting worse!)

Multiprocessing - Related

- PyMPI, MPI4Py
  - Message Passing Interface (MPI) is a very old API for multiprocessing
  - These let you use the API in python
- Global Arrays
  - Partitioned Global Addressing Space (PGAS) is a way of thinking about and writing distributed codes.
  - Global Arrays is one such implementation
  - Global Arrays seems to have some Python facing API

Threading

- Standard module
- "Parallelism" via threads
- Very similar to multiprocessing
    - Thread objects, locks, et cetera.
- Pros:
    - API is fairly standard for a task-based parallelism (create process objects, start, and join them; locks, pipe, semaphore)
- Cons:
    - Not concurrent! Bound by GIL.
    - All work and no gain.
- Apparently used for more I/O parallelism...

Threading - Terse Example

```
# Create threads
threads = [
  Thread( target=work_function, args=(work, result_queue) )
  for work in worklist
]
# Start threads
for thread in threads:
    thread.start()
# Join threads
for thread in threads:
    thread.join()
```

Threading - Benchmark Results

- Fibonacci
  - Iterative:
    - 0.0079 seconds
  - Recursive:
    - 176.43 seconds
  -

Threading - Related

- Any kind of asynchronous library is sure to work similarly to the threading module.
    - Probably about a billion of these, primarily used for web services

PyCUDA

- Non-standard module
- Write CUDA kernels in as strings, compile during runtime, and execute.
    - Very similar to how native OpenCL works (and PyOpenCL).
- Pros:
    - Can utilize very powerful hardware in a manner similar (if not identical) to the native API.
    - Spoiler alert: Very fast
- Cons:
    - Single Instruction Multiple Data (SIMD) paradigm limits its application to (essentially) array operations
    - Lots of setup that is confusing, unintuitive, and that can have a performance impact

PyCUDA - Terse Example

```
mod = SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b, int N){
  const int i = threadIdx.x;
  if( i < N )
    dest[i] = a[i] * b[i];
}
""")
# Compile CUDA function
multiply_them = mod.get_function("multiply_them")
# Create Numpy arrays for input and out
a = numpy.array( data_a ).astype(numpy.float32)
b = numpy.array( data_b ).astype(numpy.float32)
dest = numpy.zeros_like(a)
# Caluclate blocksize and grid_size
block_size = 400
grid_size = int( math.ceil(N/float(block_size)) )
# Call CUDA function
multiply_them(
  driver.Out(dest), driver.In(a), driver.In(b), numpy.int32( a.shape[0] ),
  block=(400,1,1), grid=(grid_size), 1)
)
```

PyCUDA - Benchmark Results

- Jacobi
  - 0.0507 seconds (10693x faster CPython, 17.88x faster than C!)
  - 98.619 Giga FLOPS/second

PyCUDA - Related

- PyOpenCL
  - Same but with OpenCL codes
  - Developed by same person/group
- PyChapel
  - Chapel is a high performance programming language from Cray.
  - PyChapel lets you write/use Chapel code and make calls to it from Python

Cython

- Broadly, a Python/Cython-to-C compiler
- Can compiler most Python code to a C-API implemented module
- Has Cython language for writing
- Pros:
  - Essentially compiled python
  - Seems to port relatively easily (some minor build process required)
- Cons
  - Does not work in interpreters that dont implement the CPython C-API (limited PyPy support)

Cython - Benchmark Results

- Fibonacci
  - Iterative
    - 8.70227813721e-05s
  - Recursive
    - 15.0813598633s
- Jacobi
  - 221.64 seconds (2.2x faster than CPython)
  - 22.55 Mega FLOPS/second

Cython - Related

- PyFort
  - Very similar.
  - Write Python-y Fortran that gets compiled and is callable from Python.
  - Pervasive in SciPy (~25% of project).
- SWIG
  - More for creating interfaces between existing code in a language.
- Grumpy
  - Python to Go transpiler + runtime
- Rust?

Numba

- Annotation system for Python (and maybe an interpreter?)
- Annotate loops with `@jit` or `@vectorize`
  - JIT compiler lowers into LLVM, optimizes, and then to compiles machine code during runtime.
- Pros:
  - Easy to use (on paper). Just annotate
- Cons:
  - Not actually all that easy to use.
    - Confusing type system that does not seem to terminate.
    - When are things arrays? When are they not?
    - Sometimes need to modify code to make work at all.

Numba - Terse Example

```
#Tell numba to JIT foo, infer types
@jit
def foo( a,b ):
    return a+b

# Tell numba to JIT foo, specifically for these types
@jit([ int32(int32,int32),
       float32(float32,float32),
       float64(float64,float64)])
def bar( a, b ):
  return a-b
```

Numba - Benchmark Results

- Fibonacci
  - Iterative
    - 0.082 seconds (25% slower than CPython baseline)
  - Recursive
    - 1.24985098839 (47x faster than CPython baseline)
- Jacobi
  - 542.145 seconds (11% slower than CPython baseline)

Conclusion

- No magic wand.
  - This is the norm, and is not surprising.
- Quite usable.
  - Writing high performance *always* requires fairly large code modifications.
  - Notably easier to modify in python than with than other languages, like C/C++
- If you need ***peak*** performance, Python won't be your main application.
  - But Python will work for most people systems
    - You Aren't Google - Ozan Onay https://blog.bradfieldcs.com/you-are-not-google-84912cf44afb
  - Python can be used in other places, such as orchestration of your application(s)
    - Either as a bash replacement
    - Call low level operations (see PyCUDA)