Python for HPC – Day 3

April 14, 2022

Presented by:
Nicholas A. Danes, PhD
Computational Scientist
Cyber Infrastructure & Advanced Research Computing (ITS)
Goals

• Spotlight advanced Python capacities for scientific computing
  • Profiling & Optimizing Python code
  • Exploring when (and when not to use) NumPy compared to:
    • Pure Python
    • Cython
    • Numba
  • Other ways to optimize your code
  • Shared vs Distributed memory computing
  • Mpi4py
  • Petsc4py
Note: Optimization before Parallelization!

• “Premature optimization is the root of all evil” – Donald Knuth

• Often, writing your code to run as fast as possible (within reason) with a single core is necessary before thinking about parallelization.

• We will explore optimize with a simple Python code for a single core next!
How to profile Python code

• cProfile
  • Gives you a breakdown of all functions’ runtime in a code
  • Multiple ways to use it:
    • Call it in the command line:
      
      ```
      $ python -m cProfile myscript.py
      ```
    • Call it in another script:
      
      ```
      import cProfile
      cProfile.run("mycode.main()")
      ```

• Other options: lineprofiler, timeit, pstats

References:
https://towardsdatascience.com/how-to-profile-your-code-in-python-e70c834fad89
https://github.com/pyutils/line_profiler
A starting point for optimization: Writing an ODE solver

Consider the initial value problem of the form:

\[ y'(t) = f(t, y) \]
\[ y(t_0) = y_0 \]

which can numerically solved using Heun’s Method:

\[ \hat{y}[i+1] = y[i] + h f(t[i], y[i]) \]
\[ y[i+1] = y[i] + \frac{h}{2} \left( f(t[i], y[i]) + f(t[i+1], \hat{y}[i+1]) \right) \]

Where \( h \) is the time step size, \( i \) is the time step index, and \( \hat{y} \) denotes the intermediate solution. Let’s use this problem to see how to optimize writing scientific code for Python!
Demo: Profiling multiple versions of our ODE code

- Pure Python
  - Surprisingly Performant!
- NumPy only
  - Performs poorly due to lack of vectorization
- NumPy + Numba
  - https://numba.pydata.org/
  - Numba is a JIT-compiler that converts a subset of NumPy + Python code into fast machine code
  - Performs better than NumPy
- NumPy + Cython
  - https://cython.org/
  - Cython effectively allows one to write static-typed code in Python/”Cython”, which is parsed into C and compiled into a Python module.
Other ways to think about optimizing scientific Python code

• NumPy
  • Check for vectorization possibilities!
    • Use \( v[0:n] = \text{np.sin}(x[0:n]) \) instead of
      
      ```python
      for i in range(0,n):
        v[i] = np.sin( x[i] )
      ```
  • Numba
    • Explore when you can use the JIT compiler
      • *Will not be compatible with non-NumPy/Python functions and some NumPy/Python functions*

• When possible, use sparse data structures!
  • SciPy provides these!

• If you have to write a loop, use another language and/or wrap it to your Python code
  • Cython
  • F2py (Fortran) - [https://numpy.org/doc/stable/f2py/usage.html](https://numpy.org/doc/stable/f2py/usage.html)
  • Pybind11 (C++) - [https://github.com/pybind/pybind11](https://github.com/pybind/pybind11)
Parallel Programming in Python

• Shared vs Distributed Memory Programming
  • Shared (e.g. OpenMP)
    • All CPU cores have access to the same pool of memory
    • Typically, all CPU cores are on the same CPU node
    • Ideal for multi-threaded loops
  • Distributed-memory program (e.g. MPI)
    • Each CPU core is given access to a specific pool of memory, which may or may not be shared
    • A “communicator” designates how each CPU core can talk to another CPU core
    • CPU cores do not have to live on the same CPU node
Python and the GIL: A constraint on shared memory programming

• Python Global Interpreter Lock (GIL)
  • A mechanism with Python which allows only one CPU thread to use the Python interpreter
  • The GIL addressed the problem of memory management for Python programs.
  • Releasing the GIL can cause memory leaks if not managed correctly.

• Solutions:
  • Use multiprocessing instead of multithreading
    • Each process gets its own Python interpreter and memory space
    • Module options: mpi4py, multiprocessing
  • Use a different interpreter
  • Use Cython to release the GIL to allow multithreading within subroutines

Reference: https://realpython.com/python-gil/
A brief introduction to mpi4py:

• mpi4py provides bindings for the Message Passing Interface (MPI) for Python

• MPI is a library that provides the ability for processors to communicate and send/receive data to one another, while simultaneously running concurrently in a computation

• Most parallel scientific codes use MPI for their parallelism

• Some codes allow a “hybrid” approach which allows one to combine MPI (multiprocessing) and OpenMP (multithreading) into a single code

Reference: [https://realpython.com/python-gil/](https://realpython.com/python-gil/)
A brief introduction to petsc4py:

• PETSc (Portable Extensible Toolkit for Scientific Computation) is a software suite of data structures, solvers and other routines for scalable (e.g. parallel) scientific computing

• Petsc4py is a C-wrapped library to use PETSc in Python
  • Compatible with mpi4py for distributed memory communication

• Used in some popular scientific packages such as FEniCS

Demo: Setup mpi4py and petsc4py in a conda env

• **Note:** For best performance, do NOT use conda’s binary package version of mpi4py

• To setup mpi4py to use the system’s MPI see: [https://researchcomputing.princeton.edu/mpi4py](https://researchcomputing.princeton.edu/mpi4py)

• We will showcase a demo code that uses both petsc4py and mpi4py
  • MPI “Hello World”: [https://researchcomputing.princeton.edu/mpi4py](https://researchcomputing.princeton.edu/mpi4py)
  • petsc4py 2D Poisson: [https://gitlab.com/petsc/petsc/-/blob/master/src/binding/petsc4py/demo/poisson2d/poisson2d.py](https://gitlab.com/petsc/petsc/-/blob/master/src/binding/petsc4py/demo/poisson2d/poisson2d.py)
Further Resources

Python Parallel Processing
https://wiki.python.org/moin/ParallelProcessing

Parallel Programming with MPI for Python

Intro to F2Py
https://www2.atmos.umd.edu/~dkleist/docs/pythonTraining/Slides/F2Py_SSSO.pdf