Efficient use of Python on the clusters

Ariel Lozano and Nicolas Potvin

CÉCI training

November 16, 2023

Excercise to experiment: profile a python code

- From a CECI cluster copy this folder to your home directory

```
cp -r /CECI/proj/training/python4hpc ~/
```

- Follow the instructions on the readme file

```
~/python4hpc/exercises/README.md
```

You will find a Python implementation to solve the 2D diffusion equation

Numpy library

- Provides a new kind of `array` datatype
- Contains methods for fast operations on entire arrays avoiding to define (inneficient) explicit loops
- They are basically wrappers to compiled C/Fortran/C++ code
- Their methods runs almost as fast as C compiled code
- It is the foundation of many other higher-level numerical tools
- Compares to MATLAB in functionality
- Check the use of slice indexing to iterate

```
>>> import numpy as np
>>> a = np.array([[ 5, 1 ,3],
                  [ 1, 1 ,1],
                  [ 1, 2 ,1]])
>>> b = np.array([1, 2, 3])
>>> c = a.dot(b)
array([16, 6, 8])
```

Python Bindings

We saw that interfacing python with compiled code can provide huge performance gains. There are two main approaches to achieve this:

- Compile python (or python-like) code
- Link python to use existing libraries written in other languages

Compile Python

- Just in time (JIT) compilers: compile and run a python code in real time
  - Numba: jit compiler supporting numpy code
- Ahead of time (AOT) compilers: creation of a compiled library in your machine (this would provide what is called a *binding)*
  - *Cython: compile a python-like C code or a pure C library*
  - *f2py: tool part of numpy project allowing to compile and wrap Fortran code*

Compile Python: Fibonacci example

The Fibonacci series is defined by the recurrence relation

$$F_n = F_{n-1} + F_{n-2} \tag{1}$$

starting with $F_0 = 0$ and $F_1 = 1$.
A basic pure python implementaion:

```
def fibonacci(num):
    fn = 0
    fn1 = 1
    while num-1:
        fn, fn1 = fn1, fn + fn1
        num -= 1
    return fn1


if __name__ == "__main__":

    print(fibonacci(15))
```

Compile python: Cython

You must annotate your code using a new syntax in between python and C.
Example `fibonacci` function in cython[1]

```
def fibonacci(int num):
    cdef int fn
    cdef int fn1
    fn = 0
    fn1 = 1
    while num-1:
        fn, fn1 = fn1, fn + fn1
        num -= 1
    return fn1
```

To build it is required a sort of makefile, typically called `setup.py`

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Build import cythonize

setup(ext_modules = cythonize("fibolib.pyx"))
```

[1] code files on `python4hpc/examples/compiling/fibo-cython`

Compile python: Cython

The build will produce a binary `.so` object for the library

```
$ python setup.py build_ext --inplace
```

Having this lib on the same directory, it can be imported as a module on a pure python code

```python
from fibolib import fibonacci

print(fibonacci(15))
```

Python bindings: C libraries

- Cython allows also to wrap C libraries to provide bindings for Python
- Check the example in `python4hpc/examples/compiling/fibo-wrap-c` to see how wrapping works for a C function providing the n_{th} Fibonacci number.
- Steps for building and running the example:

```
$ make
$ python fibonacci.py
The 15th Fibonacci number is: 610
```

Python Bindings: f2py example

- To wrap Fortran code the `f2py` tool from numpy provides a straighforward approach[2]

```fortran
function fibonacci(n)
    implicit none
    integer, intent(in) :: n
    integer :: fibonacci, fseries(0:n), i
    fseries(0) = 0
    fseries(1) = 1

    do i = 2, n
      fseries(i) = fseries(i - 1) + fseries(i - 2)
    end do
    fibonacci = fseries(n)
end function fibonacci
```

```python
import fibolib

print(fibolib.fibonacci(15))
```

```
$ f2py -c -m fibolib fibolib.f90
```

```
$ python fibonacci.py
  610
```

[2] code files on `python4hpc/examples/compiling/fibo-fortran`