Efficient use of Python on the clusters

Ariel Lozano

CÉCI training

November 25, 2020

Outline

- Analyze our code with profiling tools:
  - cpu: `cProfile`, `line_profiler`, `kernprof`
  - memory: `memory_profiler`, `mprof`
- How to make a more efficient use of hardware internals?
  - Numpy and Scipy ecosystem (mainly wrappers to C/Fortran compiled code)
  - binding to compiled code: interfaces between python and compiled modules
  - compiling: tools to compile python code
  - parallelism: overview of modules to exploit multicores

Sieve of eratostenes

Algorithm to find all prime numbers up to any given limit.

Ex: Find all the prime numbers less than or equal to 25:

- 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

Cross out every number displaced by 2 after 2 up to the limit:

- 2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 ~~14~~ 15 ~~16~~ 17 ~~18~~ 19 ~~20~~ 21 ~~22~~ 23 ~~24~~ 25

Move to next *n* non crossed, cross out each non crossed number displaced by *n*:

- 2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~ 17 ~~18~~ 19 ~~20~~ ~~21~~ ~~22~~ 23 ~~24~~ 25
- 2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~ 17 ~~18~~ 19 ~~20~~ ~~21~~ ~~22~~ 23 ~~24~~ ~~25~~

The remaining numbers non crossed in the list are all the primes below *limit*.
Trivial optimization: jump directly to n^2 to start crossing out. Then, n must loop only up to $\sqrt{limit}$.

Simple pure python implementation

```python
def primes_upto(limit):
    sieve = [False] * 2 + [True] * (limit - 1)
    for n in range(2, int(limit**0.5 + 1)):
        if sieve[n]:
            i = n**2
            while i < limit+1:
                sieve[i] = False
                i += n
    return [i for i, prime in enumerate(sieve) if prime]


primes = primes_upto(25)
print(primes)
```

```
$ python3 sieve01_print.py

[2, 3, 5, 7, 11, 13, 17, 19, 23]
```

Measuring running time

Computing primes up to 30M:

- linux `time` command

```
$ time python3 sieve01.py

real    0m10.419s
user    0m10.192s
sys     0m0.217s
```

- using `timeit` module to average several runs

```
$ python3 -m timeit -n 3 -r 3 -s "import sieve01" "sieve01.primes_upto(30000000)"

3 loops, best of 3: 10.2 sec per loop
```

CPU profiling: timing functions

`cProfile`: built-in profiling tool in the standard library. It hooks into the virtual machine to measure the time taken to run every function that it sees.

```
$ python3 -m cProfile -s cumulative sieve01.py
        5 function calls in 10.859 seconds

  Ordered by: cumulative time

  ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       1    0.000    0.000   10.859   10.859 {built-in method builtins.exec}
       1    0.087    0.087   10.859   10.859 sieve01.py:3(<module>)
       1    9.447    9.447   10.772   10.772 sieve01.py:3(primes_upto)
       1    1.325    1.325    1.325    1.325 sieve01.py:11(<listcomp>)
       1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

- For big codes use a visualization tool as snakeviz

```
$ python3 -m cProfile -o profile.stats sieve01.py
$ snakeviz profile.stats
```

CPU profiling: line by line details of a function

`line_profiler`: profiling individual functions on a line-by-line basis, **big overhead** introduced. We must add the `@profile` decorator on the function to be analyzed.

```
@profile
def primes_upto(limit):
    sieve = [False] * 2 + [True] * (limit - 1)
    for n in range(2, int(limit**0.5 + 1)):
        if sieve[n]:
            i = n**2
            while i < limit+1:
                sieve[i] = False
                i += n
    return [i for i, prime in enumerate(sieve) if prime]


primes = primes_upto(30000000)
```

Then, we run the code with the `kernprof` tool provided by the package.

CPU profiling: line by line details of a function

```
 $ kernprof -l -v sieve01_prof.py
Wrote profile results to sieve01_prof.py.lprof
Timer unit: 1e-06 s

Total time: 101.025 s
File: sieve01_prof.py
Function: primes_upto at line 2

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     2                                           @profile
     3                                           def primes_upto(limit):
     4         1     229258.0 229258.0      0.3      sieve = [False] * 2 + [True] * (limit - 1)
     5      5477       2466.0      0.5      0.0      for n in range(2, int(limit**0.5 + 1)):
     6      5476       2578.0      0.5      0.0          if sieve[n]:
     7       723        855.0      1.2      0.0              i = n**2
     8  70634832   28295172.0      0.4     32.4              while i < limit+1:
     9  70634109   29280104.0      0.4     33.5                  sieve[i] = False
    10  70634109   26771040.0      0.4     30.7                  i += n
    11         1    2740062.0 2740062.0      3.1      return [i for i, prime in enumerate(sieve) if prime]
```

`% Time` is relative to lines on the function, not to total run time

Memory profiling: line by line details of a function

`memory_profiler`: module to measure memory usage on a line-by-line basis, runs will be slower than `line_profiler`. Is also required the `@profile` decorator on the function to analyze.

```
 $  python3 -m memory_profiler sieve01_prof.py
 Filename: sieve01_prof.py

Line #    Mem usage    Increment   Line Contents
================================================
     2   32.715 MiB    0.000 MiB   @profile
     3                             def primes_upto(limit):
     4  261.703 MiB  228.988 MiB       sieve = [False] * 2 + [True] * (limit - 1)
     5  261.703 MiB    0.000 MiB       for n in range(2, int(limit**0.5 + 1)):
     6  261.703 MiB    0.000 MiB           if sieve[n]:
     7  261.703 MiB    0.000 MiB               i = n**2
     8  261.703 MiB    0.000 MiB               while i < limit+1:
     9  261.703 MiB    0.000 MiB                   sieve[i] = False
    10  261.703 MiB    0.000 MiB                   i += n
    11                                 return [i for i, prime in enumerate(sieve) if prime]
```

Memory profiling: line by line details of a function

Why are 228 MB allocated on this line?

```
    4   261.703 MiB   228.988 MiB         sieve = [False] * 2 + [True] * (limit - 1)
```

- In a Python list each boolean variable has a size of 8 bytes. The standard for a C `long int` in 64-bits.
- We are creating a list with 30 million elements.
- Doing the math: $\frac{30E6*8\ B}{1024*1024} = 228.881\ MB$

Remarks:

- Memory line by line analysis introduces an even bigger overhead, run can be up to 100x slower
- We can miss information due to many memory operations taking place on a single line

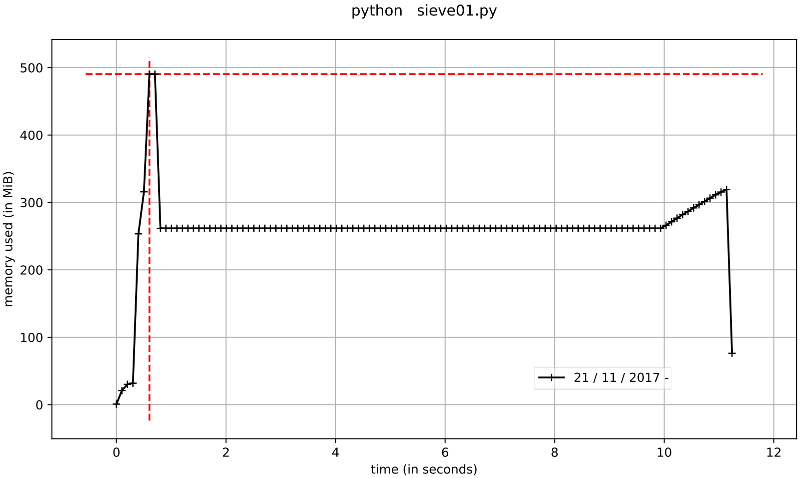
Memory profiling: analyzing the whole run vs time

- The `memory_profiler` package provides the `mprof` tool to analyze and visualize the memory usage as a function of time
- It has a very minor impact on the running time
- Usage:

```
$ mprof run --python python3 mycode.py
$ mprof plot
```

Memory profiling: analyzing the whole run vs time

```
$ mprof run --python python3 sieve01.py
$ mprof plot
```



Memory profiling: analyzing the whole run vs time

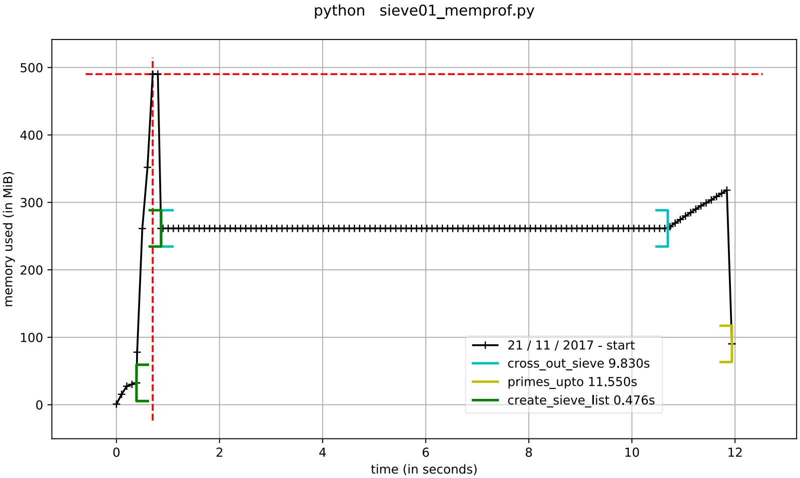
We can add a `@profile` decorator and `profile.timestamp()` labels to introduce details in the analysis

```
@profile
def primes_upto(limit):
    with profile.timestamp("create_sieve_list"):
        sieve = [False] * 2 + [True] * (limit - 1)
    with profile.timestamp("cross_out_sieve"):
        for n in range(2, int(limit**0.5 + 1)):
            if sieve[n]:
                i = n**2
                while i < limit+1:
                    sieve[i] = False
                    i += n
    return [i for i, prime in enumerate(sieve) if prime]

primes = primes_upto(30000000)
```

Memory profiling: analyzing the whole run vs time

```
$ mprof run --python python3 sieve01_memprof.py
$ mprof plot
```



Memory profiling: analyzing the whole run vs time

Why the 500 MB peak during the sieve list creation?

- Experimenting with the `mprof` tool can be verified that:

```
sieve = [False] * 2 + [True] * (limit - 1)
```

- is actually equivalent to something like:

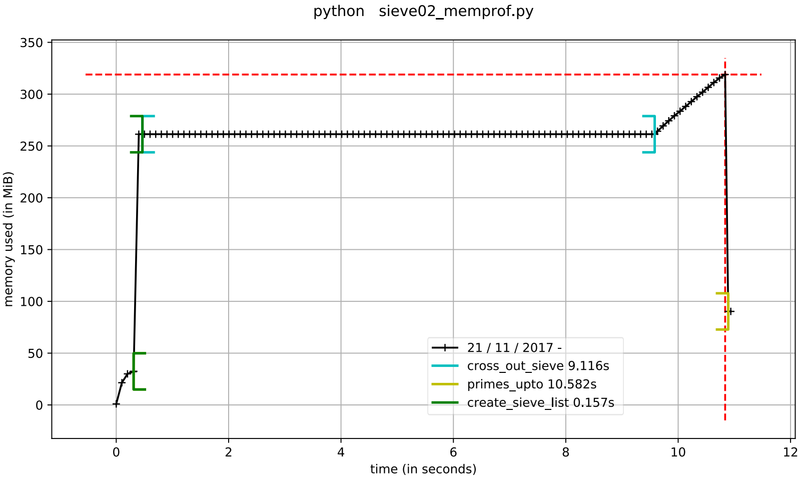
```
sieve1 = [False] * 2
sieve2 = [True] * (limit - 1)
sieve = sieve1 + sieve2
del sieve1
del sieve2
```

- is allocated temporarily an extra $\approx 30E6$ boolean list !!
- We can try to replace with:

```
sieve = [True] * (limit + 1)
sieve[0] = False
sieve[1] = False
```

Memory profiling: analyzing the whole run vs time

```
$ mprof run --python python3 sieve02_memprof.py
$ mprof plot
```



Excercise to experiment: profile a python code

Python implementation of the 2D diffusion equation

- From Lemaitre3 or Dragon2 copy this folder to your home directory

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

- Create the virtualenv and follow the **Hands on: First part** as explained on

```
~/python4hpc/excercises/README.md
```

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

```
>>> 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])
```

Numpy library: sieve revisited

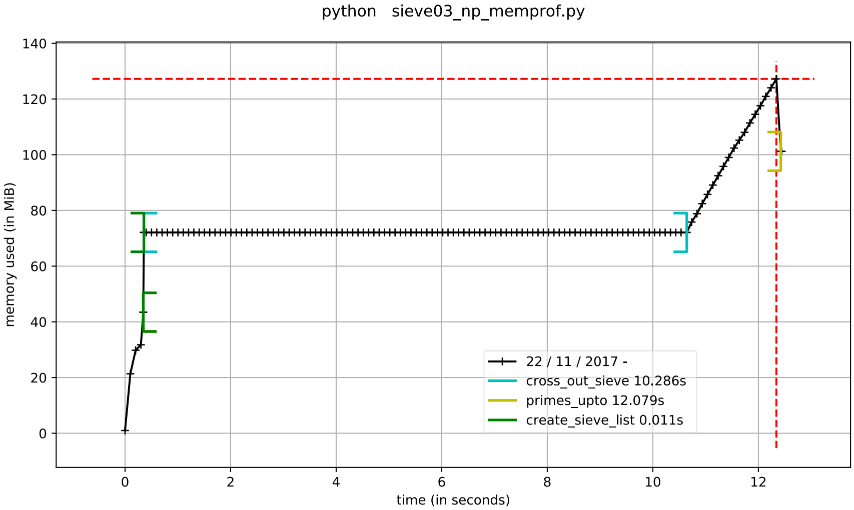
We replace the sieve list with a Numpy boolean array:

```
import numpy as np

def primes_upto(limit):
    sieve = np.ones(limit + 1, dtype=np.bool)
    sieve[0] = False
    sieve[1] = False
    for n in range(2, int(limit**0.5 + 1)):
        if sieve[n]:
            i = n**2
            while i < limit+1:
                sieve[i] = False
                i += n
    return [i for i, prime in enumerate(sieve) if prime]


primes = primes_upto(30000000)
```

Numpy library: sieve revisited



- In a Numpy array each boolean has a size of 1 byte
- Math now: $\frac{30E6 * 1\ B}{1024 * 1024} = 28.61\ MB$

Numpy library: sieve revisited

- Timing did not improve with Numpy array and same loop
- Full Numpy solution using slice indexing to iterate:

```
import numpy as np

def primes_upto(limit):
    sieve = np.ones(limit + 1, dtype=np.bool)
    sieve[0] = False
    sieve[1] = False
    for n in range(2, int(limit**0.5 + 1)):
        if sieve[n]:
            sieve[n**2::n] = 0
    return np.nonzero(sieve)[0]
```

```
 $ time python3 sieve04_np.py
real    0m0.552s
user    0m0.518s
sys     0m0.033s
```

22x gain in time!!

Numpy library: sieve line by line profiling

```
 $  kernprof -l -v sieve04_np_prof.py
Wrote profile results to sieve04_np_prof.py.lprof
Timer unit: 1e-06 s

Total time: 0.482723 s
File: sieve04_np_prof.py
Function: primes_upto at line 3

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     3                                           @profile
     4                                           def primes_upto(limit):
     5         1         8785   8785.0      1.8      sieve = np.ones(limit + 1, dtype=np.bool)
     6         1            5      5.0      0.0      sieve[0] = False
     7         1            0      0.0      0.0      sieve[1] = False
     8      5477         2796      0.5      0.6      for n in range(2, int(limit**0.5 + 1)):
     9      5476         3119      0.6      0.6          if sieve[n]:
    10       723       420784    582.0     87.2              sieve[n**2::n] = 0
    11         1        47234  47234.0      9.8      return np.nonzero(sieve)[0]
```

Numpy library: sieve line by line profiling

- `line_profiler` helps to understand the massive gain
- Pure python solution:

```
 6       5476       2362    0.4       0.0   if sieve[n]:
 7        723        680    0.9       0.0       i = n**2
 8   70634832   28740579    0.4      28.4       while i < limit+1:
 9   70634109   33142484    0.5      32.8           sieve[i] = False
10   70634109   26776815    0.4      26.5           i += n
```

- Full Numpy solution:

```
 9       5476       3119    0.6       0.6   if sieve[n]:
10        723     420784  582.0      87.2       sieve[n**2::n] = 0
```

- The loops to cross out the sieve are fully performed by lower level implementations in Numpy
- Time and memory usage is the same as best pure C or Fortran implementations!

CPU and Memory profiling: summary

- Line-by-line profiling introduces a huge overhead, they must be used reducing the problem size and for specific functions detected as bottlenecks
- The `mprof` tool is very dynamic, *timestammping* in a smart way can be used both as a fast CPU and Memory profiler
- The `cProfile` dumps are great to detect bottlenecks on big projects, but a visualization tool is almost mandatory. Explore the KCachegrind package, usual workflow:

```
$ python -m cProfile -o prof.out sieve02.py
$ pyprof2calltree -i prof.out -k
```

Numpy library: SciPy ecosystem

Collection of open source software for scientific computing in Python

- Core packages:
  - NumPy: the fundamental package for numerical computation
  - SciPy library: collection of numerical algorithms and domain-specific toolboxes, including signal processing, fourier transforms, clustering, optimization, statistics...
  - Matplotlib: a mature plotting package, provides publication-quality 2D plotting as well as basic 3D plotting
- Data and computation:
  - pandas: providing high-performance, easy to use data structures (similar to R)
  - SymPy: symbolic mathematics and computer algebra
  - scikit-image: algorithms for image processing
  - scikit-learn: algorithms and tools for machine learning
  - h5py and PyTables: can both access data stored in the HDF5 format

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
  - Pypy: jit compiler for non-numpy code
- Ahead of time (AOT) compilers: creation of a compiled static library in your machine
  - Cython: the most popular, compile a python-like C code
  - Pythran: automatic Python-to-C++ converter and compiler compatible with numpy

Compile Python: JIT

- Pypy: We can directly run the original `sieve01.py` with `pypy`
- Numba: We just need to decorate the function we wish to compile

```
from numba import jit

@jit
def primes_upto(limit):
    sieve = [False] * 2 + [True] * (limit - 1)
    for n in range(2, int(limit**0.5 + 1)):
        if sieve[n]:
            i = n**2
            while i < limit+1:
                sieve[i] = False
                i += n
    return [i for i, prime in enumerate(sieve) if prime]


primes = primes_upto(30000000)
```

Compile Python: JIT remarks

- JIT compilers offers some nice speedups with very little manual intervention
- But the more we rely on a tool to automatically optimize we are rapidly bounded on what can be improved
- Pypy is not compatible with numpy code
- Numba seems a quite promising tool and it’s numpy compatible
- You might be happy with what you obtain with very low effort, is up to your problem and how many times you are going to run your code

Compile python: Cython

You must annotate your code using a new syntax in between python and C.

Example sieve `primes_upto` function

```
def primes_upto(int limit):
    cdef int n, i
    cdef int prime
    sieve = [True]*limit
    for n in range(2, int(limit**0.5 + 1)):
        if sieve[n]:
            i = n**2
            while i < limit:
                sieve[i] = False
                i += n
    return [i for i, prime in enumerate(sieve) if prime]
```

After compiling, it can be imported in a pure python code

```
from sievelib import primes_upto

primes = primes_upto(30000000)
```

Compile python: Cython

It requires to create a sort of makefile, called typically `setup.py`, there is a working example in `example/compiling/cython`

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

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

To use the resulting module, built as a binary `.so` file, by a python script on the same directory

```
$ python setup.py build_ext --inplace
$ time python sieve01.py

real    0m2.663s
user    0m2.552s
sys     0m0.080s
```

Compile python: Pythran

It requires annotations for the type information of the function to compile

```python
#pythran export primes_upto(int)
def primes_upto(limit):
    sieve = [True]*limit
    for n in range(2, int(limit**0.5 + 1)):
        if sieve[n]:
            i = n**2
            while i < limit:
                sieve[i] = False
                i += n
    return [i for i, prime in enumerate(sieve) if prime]
```

To compile the `.so` file

```
$ pythran sievelib.py
$ time python3 sieve01.py

real    0m0.512s
user    0m0.460s
sys     0m0.051s
```

Compile python: OpenMP support

- Both Cython and Pythran provide the possibility to produce OpenMP code
- This allows to use under the hood more cores in a multicore machine
- In Cython this is enabled by using special operators (i.e. `prange` instead `range`) and compiling with the `-fopenmp` flag
- In Pythran we annotate the python code to create a parallel region similar to original OpenMP usage in C

Python bindings: C libraries

- Cython allows also to wrap C libraries to provide bindings for Python
- Check the example in `compiling/fib-wrap-c` to see how wrapping works for a C function providing the n_{th} Fibonacci number.

```
$ make
$ make test
$ python test.py
The 10th Fibonacci number is: 55
```

Python Bindings: f2py example

- To wrap Fortran code the `f2py` tool provides a more straighforward approach to do so

```fortran
subroutine foo(a)
    integer a
    print*, "Hello from Fortran!"
    print*, "a=",a
end
```

```python
import hello

hello.foo(10)
```

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

```
$ python call_fhello.py
 Hello from Fortran!
 a=           10
```

Compiled Python

- cython: C-Extensions for Python
  - optimising and static compiler
  - can compile Python code and Cython language
  - can compile Python with Numpy code
  - can do bindings with C code
- Pypy: Just-in-time compiler
  - sometimes less memory hungry than Cython
  - not fully compliant with Python with Numpy code
- Numba: a compiler specialized for numpy code using the LLVM compiler
- Pythran: compiler for both numpy and non-numpy code. Takes advantage of multi-cores and single instruction multiple data (SIMD) units
- All, except `pypy` requires to modify or decorate the original python code

Parallel processing

- `multiprocessing` module
  - allows to use process- and thread-based parallel processing
  - for multi-process can be non trivial to share memory among them
- `joblib` module
  - desinged for scientific use in mind
  - optimized for numpy arrays
  - focused on embarrasingly parallel kind of problems
- `mpi4py`
  - Python bindings to the MPI-1/2/3 interfaces
  - if *you know* MPI on C/Fortran *you already know* mpi4py
  - can make use equivalently of multiple cores on a single-machine or distributed
  - each process has a separate address space, no possibility to share memory between them
  - we covered it in the MPI session

Excercise to experiment: compile python code

- Follow the **Hands on: Second part** as explained on

```
~/python4hpc/excercises/README.md
```

Further references and training on the topic

- High Performance Python - 2nd Ed by By Micha Gorelick and Ian Ozsvald
- Python in HPC Tutorial: `https://github.com/pyHPC/pyhpc-tutorial`
- PRACE Sponsored Online Course: Python in High Performance Computing