High Performance Computing with Python

Memory management and GIL

R. Sarmiento, V. Karakasis, T. Manitaras and T. Robinson
ETHZürich / CSCS
Lugano - 11.11.2019

ETH zürich
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

HPC

- PetaFLOPS
- Exascale computing
- Scaling to a larger number of nodes

HPC

- PetaFLOPS
- Exascale computing
- Scaling to a larger number of nodes
- CPU features
- Memory management
- Optimized use of CPU caches
- Accelerators
- Efficient IO
- OpenMP and MPI

Python

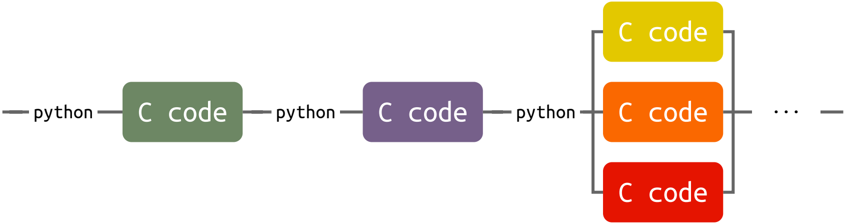
- It’s pretty cool

Python

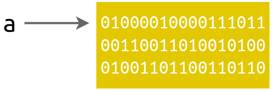
- It’s pretty cool
- It’s fairly easy to glue it to other languages like C and Fortran
- Most of it’s operations can be overloaded

Python

- It’s pretty cool
- It’s fairly easy to glue it to other languages like C and Fortran
- Most of it’s operations can be overloaded

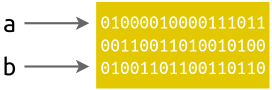


Memory management: Reference counting and garbage collection



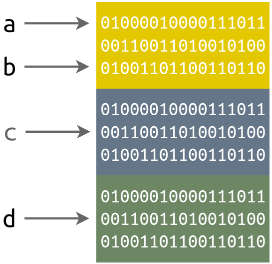

```
a = np.random.random((m, m))
```

Memory management: Reference counting and garbage collection




```
a = np.random.random((m, m))
b = a.T  # increases the reference count
```

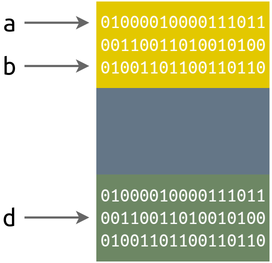
Memory management: Reference counting and garbage collection




```
a = np.random.random((m, m))
b = a.T  # increases the reference count

c = np.random.random((m, m))
d = np.random.random((m, m))
```

Memory management: Reference counting and garbage collection

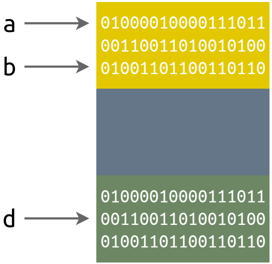



```
a = np.random.random((m, m))
b = a.T   # increases the reference count

c = np.random.random((m, m))
d = np.random.random((m, m))

del c     # set reference count to 0
```

Memory management: Reference counting and garbage collection

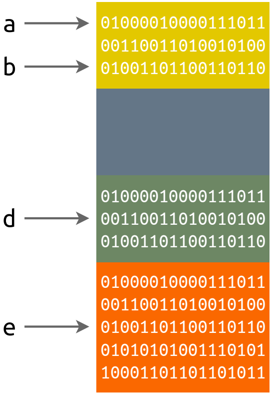



```
a = np.random.random((m, m))
b = a.T   # increases the reference count

c = np.random.random((m, m))
d = np.random.random((m, m))

c = 1     # set reference count to 0
```

Memory management: Reference counting and garbage collection



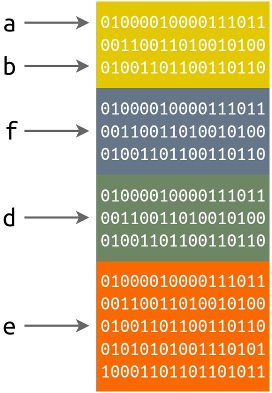

```python
a = np.random.random((m, m))
b = a.T   # increases the reference count

c = np.random.random((m, m))
d = np.random.random((m, m))

del c     # set reference count to 0

e = np.random.random((n, n))   # n > m
```

Memory management: Reference counting and garbage collection




```python
a = np.random.random((m, m))
b = a.T  # increases the reference count

c = np.random.random((m, m))
d = np.random.random((m, m))

del c    # set reference count to 0

e = np.random.random((n, n))   # n > m

f = np.random.random((m, m))
```

Global interpreter lock (GIL) in CPython

A **Lock** is a mechanism for enforcing limits on access to a resource in an environment where there are many threads of execution

Global interpreter lock (GIL) in CPython

A **Lock** is a mechanism for enforcing limits on access to a resource in an environment where there are many threads of execution

- `acquire()`
- `release()`

Global interpreter lock (GIL) in CPython

```python
import threading
lock = threading.Lock()


def function1():
    for i in range(5):
        lock.acquire()
        print('Function 1 running')
        lock.release()

def function2():
    for i in range(5):
        lock.acquire()
        print('Funcion 2 running')
        lock.release()

thread_1 = threading.Thread(target=function1)
thread_2 = threading.Thread(target=function2)
thread_1.start()
thread_2.start()
thread_1.join()
thread_2.join()
```

Global interpreter lock (GIL) in CPython

- CPU bound

```
...
acquire_lock()
 // do something
release_lock()  // let other threads do something
...
```

Global interpreter lock (GIL) in CPython

- CPU bound

```
...
acquire_lock()
 // do something
release_lock()  // let other threads do something
...
```

- IO bound (waiting from OS calls)

```
...
release_lock()  // let other threads do something
 // do the io task
acquire_lock()
 // go back to the interpreter
...
```

Global interpreter lock (GIL) in CPython

```
... //some_numpy_function.c

// release the GIL
NPY_LOOP_BEGIN_THREADS

// do something

// acquire the GIL
NPY_LOOP_END_THREADS
...
```

- `cray-python`
- `module load cray-python/<version>`
- Uses `cray-libsci` as backend for NumPy

```
>>> import numpy as np
>>> np.show_config()
openblas_info:
    libraries = ['sci_gnu_mp', 'sci_gnu_mp']
    library_dirs = ['/opt/cray/pe/libsci/default/GNU/7.1/x86_skylake/lib']
    language = c
    define_macros = [('HAVE_CBLAS', None)]
blas_opt_info:
    libraries = ['sci_gnu_mp', 'sci_gnu_mp']
    library_dirs = ['/opt/cray/pe/libsci/default/GNU/7.1/x86_skylake/lib']
    language = c
    define_macros = [('HAVE_CBLAS', None)]
...
```

- Uses the libraries installed in the system by Cray
- `module load PyExtensions/<cray-python-version>`
- `module load TensorFlow`
- `pip install --user <package>`

- Anaconda/Miniconda
- Needs to be installed by the user
- Uses Intel's MKL as backend for NumPy

```
>>> import numpy as np
>>> np.show_config()
blas_mkl_info:
    libraries = ['mkl_rt', 'pthread']
    library_dirs = ['/home/sarafael/software/anaconda3.6/lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['/home/sarafael/software/anaconda3.6/include']
blas_opt_info:
    libraries = ['mkl_rt', 'pthread']
    library_dirs = ['/home/sarafael/software/anaconda3.6/lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['/home/sarafael/software/anaconda3.6/include']
...
```

- Anaconda brings it's own libraries. An Anaconda installation shouldn't be mixed with Cray's modules (in general).
- `conda install -c <channel> package`
- `pip install --user <package>`

- Python Package Index (PyPI)
- `pip install --user <package>`
- In general PyPi offers binaries built without a specific target architecture to ensure their portability.
- Before installing with `pip` or `conda`, it might be a good idea to check the recommended installation in package's homepage or consider building it from sources.

Thank you for your attention!