Programming Machine Learning Algorithms for HPC

- Python
- Numpy

Dr. Pierrick Pochelu and Dr. Oscar J. Castro-Lopez

About me

Oscar J. Castro-Lopez
Postdoctoral researcher
Machine learning + HPC

Research interests:

- Machine learning, High performance computing, Software engineering.

Current projects/interests:

- Surrogate machine learning models for discrete event simulation.
- Cloud discovery using machine learning models.
- ML deployment

Python

Introduction and basic features

Python

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming.

Python’s elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

Python features

Programs written in Python are typically much shorter than equivalent C, C++, or Java programs, for several reasons:

- the high-level data types allow you to express complex operations in a single statement;
- statement grouping is done by indentation instead of beginning and ending brackets;
- no variable or argument declarations are necessary.

Python's popularity

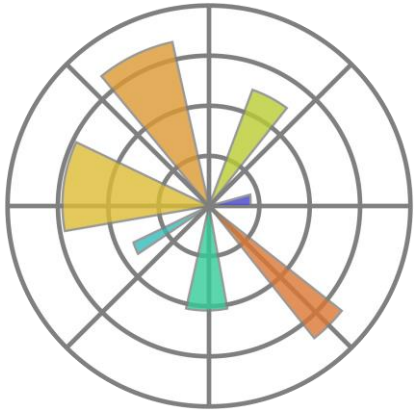
Python is *extensible*: if you know how to program in C it is easy to add a new built-in function or module to the interpreter, either to perform critical operations at maximum speed, or to link Python programs to libraries that may only be available in binary form (such as a vendor-specific graphics library).

1 on the Tiobe Index an indicator of the popularity of programming languages.

https://www.tiobe.com/tiobe-index/

Popular Python libraries

pandas



Matplotlib

NumPy

Keras

scikit learn

TensorFlow

SciPy

PyTorch

django

Flask

FastAPI

Python and HPC (part 1)

Compared to other languages Python is often considered slow:

- **Interpreted Language**: Python is an interpreted language, meaning the code is executed line by line at runtime rather than being compiled into machine code beforehand.
- **Global Interpreter Lock (GIL)**: Python uses a Global Interpreter Lock, which prevents multiple native threads from executing Python bytecodes simultaneously
- **Dynamic Typing**: Python is dynamically typed, meaning that variable types are determined at runtime.

Python and HPC (part 2)

- **Memory Management**: Python uses automatic memory management (garbage collection).
- **High-level Abstractions**: Python provides many high-level abstractions, such as list comprehensions, generators, and dynamic data types, which, while easy to use, often come with performance trade-offs compared to lower-level operations.
- **Interpreted Nature of Libraries**: Even though Python provides access to powerful libraries, some of these libraries, particularly those not written in lower-level languages (like C), may introduce additional performance bottlenecks.

GIL (Python Global Interpreter Lock)

- Lock that allows only one thread to hold the control of the Python interpreter.
- This means that only one thread can be in a state of execution at any point in time.
- The impact of the GIL isn’t visible to developers who execute single-threaded programs, but it can be a performance bottleneck in CPU-bound and multi-threaded code

Why was the GIL implemented in Python?

- Python uses reference counting for memory management. It means that objects created in Python have a reference count variable that keeps track of the number of references that point to the object.
- When this count reaches zero, the memory occupied by the object is released.

```
import sys
a = []
b = a
sys.getrefcount(a)
```

```
import sys
a = []
b = a
c = a
sys.getrefcount(a)
```

GIL test

```
import sys
a = []
b = a
sys.getrefcount(a)
```

```
import sys
a = []
b = a
c = a
sys.getrefcount(a)
```

The problem was that this reference count variable needed protection from race conditions where two threads increase or decrease its value simultaneously. This can cause the following:

- It can cause either leaked memory that is never released.
- Release the memory while a reference to that object still exists.
- This reference count variable can be kept safe by adding locks to all data structures that are shared across threads so that they are not modified inconsistently.
- But adding a lock to each object or groups of objects means multiple locks will exist which can cause another problem—Deadlocks (deadlocks can only happen if there is more than one lock) and can also decrease performance.
- The GIL is a single lock on the interpreter itself which adds a rule that execution of any Python bytecode requires acquiring the interpreter lock. This prevents deadlocks (as there is only one lock) and doesn't introduce much performance overhead. **But it effectively makes any CPU-bound Python program single-threaded**.

Why Python still has the GIL?

- Python cannot bring a change as significant as the removal of GIL without causing backward incompatibility issues.
- Discussed by the creator of Python: https://www.artima.com/weblogs/viewpost.jsp?thread=214235
- To be removed in future versions?

Python alternatives for HPC

- multiprocessing — Process-based parallelism: The multiprocessing package offers both local and remote concurrency, effectively side-stepping the Global Interpreter Lock by using subprocesses instead of threads.

- Using libraries: mpi4py, CUpy, pyCUDA, PyMP, Numba, Cython, etc…

- Alternative Python compilers/interpreter: Python has multiple interpreter implementations. CPython, Jython, IronPython and PyPy, written in C, Java, C# and Python respectively, are the most popular ones. The GIL exists only in the original Python implementation that is CPython.

Numpy

For speeding up linear algebra operations

What is NumPy?

- It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices).

- Additionally, contains an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation, etc.

- More information on: https://numpy.org/

Key characteristics of Numpy

- While a Python list can contain different data types within a single list, all the elements in a NumPy array should be homogeneous.
- NumPy arrays are faster and more compact than Python lists. An array consumes less memory.
- NumPy uses much less memory to store data, and it provides a mechanism of specifying the data types. This allows the code to be optimized even further.
- An **array** is a central data structure of the NumPy library. It is a grid of values, and it contains information about the raw data, how to locate an element, and how to interpret an element.

Attributes of an array

- Is usually a fixed-size container of items of the same type and size.
- The number of dimensions and items in an array is defined by its shape.
- The shape of an array is a tuple of non-negative integers that specify the sizes of each dimension.
- In NumPy, dimensions are called axes.

```
import numpy as np
a = np.array([1, 2, 3, 4, 5])
print(a)
```

Array creation

```
matrix_sm = np.arange(16).reshape((4,4))
print(matrix_sm)
```

Matrix creation

Shape and Size of an array

- Print the shape, size and number of dimension of the arrays:

```
print(f'a shape: {a.shape} - a size: {a.size} - - a
dimensions: {a.ndim}')

print(f'matrix shape: {matrix_sm.shape} - matrix size:
{matrix_sm.size} - matrix dimensions {matrix_sm.ndim}’)

print(a.dtype, matrix_sm.dtype)
```

Adding a new axis to an array

- You can use np.newaxis and np.expand_dims to increase the dimensions of your existing array.

```
# With np.newaxis
a = np.arange(5)
print(a, a.shape)

a2 = a[np.newaxis, :]
print(a2, a2.shape)

a2 = a[:, np.newaxis]
print(a2, a2.shape)
```

```
# With np.expand_dims
a = np.arange(5)
print(a, a.shape)

a2 = np.expand_dims(a, axis=0)
print(a2, a2.shape)

a2 = np.expand_dims(a, axis=1)
print(a2, a2.shape)
```

Numpy's Operations and broadcasting

- Broadcasting in NumPy allows you to perform operations between arrays of different shapes or between an array and a scalar value.

- It simplifies the process by automatically aligning and extending the smaller array to match the shape of the larger one, enabling you to perform element-wise operations without explicitly reshaping the arrays.

- This powerful feature simplifies code and makes it more concise when working with arrays of different sizes in NumPy.

Broadcasting example 1

```
matrix_sm = np.arange(16).reshape((4,4))
print(matrix_sm)

matrix_sm * 2
# Two vectors operation
row1 = np.arange(4)
print(row1)
row2 = np.arange(4) * 2
print(row2)
print(row1 + row2)
# Matrix and vectors operation
row_sm = np.arange(4)
print(row_sm)
print()
print(matrix_sm * row_sm)
```

```
# Operation matrix and row
row_sm = np.arange(5)
print(row_sm)
print()
print(matrix_sm * row_sm)
```

General Broadcasting Rules

When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing (i.e., rightmost) dimension and works its way left. Two dimensions are compatible when

- they are equal, or
- one of them is 1.

More details on:

https://numpy.org/doc/stable/user/basics.broadcasting.html

Broadcasting example 2

```
matrix_sm =
np.arange(20).reshape((5,4))
print(matrix_sm.shape)
row_sm = np.arange(5)
print(row_sm.shape)
#
matrix_sm * row_sm
#
row_sm2 = row_sm[np.newaxis, :]
print(row_sm2.shape)
print()
print(row_sm2)
#
matrix_sm * row_sm2
```

```
print(matrix_sm.shape)
print(row_sm2.shape)
#
row_sm2 = row_sm[:, np.newaxis]
print(row_sm2.shape)
print()
print(row_sm2)
#
matrix_sm * row_sm2
#
print(matrix_sm.shape)
print(row_sm2.shape)
```

Numpy operators with argument axis

- When an operation or function includes the axis argument, it is used to specify the axis along which the operation should be applied. NumPy arrays can have multiple dimensions, and the axis argument allows you to control whether an operation should be performed across rows, columns, or some other dimension.

```
matrix_sm = np.arange(16).reshape((4,4))
print(f'Matrix content: \n{matrix_sm}\n')
# matrix sum without the axis
print(f'Matrix result of sum all values: {np.sum(matrix_sm)}\n')
# matrix sum with axis = 0 - sum across columns
print(f'Matrix result of sum across columns: {np.sum(matrix_sm, axis=0)}\n')
# matrix sum with axis = 1 - sum across rows
print(f'Matrix result of sum across rows: {np.sum(matrix_sm, axis=1)}')
```

Minimal speed test Python vs Numpy

```
import numpy as np
array1 = np.arange(1000000)
array2 = np.arange(1000000)
# Python
total = np.zeros((array1.shape),
dtype=array1.dtype)
for i in range(array1.shape[0]):
    total[i] = array1[i] + array2[i]

print("Total:", total)
```

```
import numpy as np
array1 = np.arange(1000000)
array2 = np.arange(1000000)
# Python
total = array1 + array2
print("Total:", total)
```