Importare il C++ in Python velocemente

++it Italian C++ Community

www.italiancpp.org

Federico Pasqua

federico.pasqua.96@gmail.com

Python: Pro e Contro

- Interpretato
- Facile da tenere pulito e comprensibilie
- Per ogni cosa c’è un modulo pronto all’uso
- Veloce da scrivere, ottimo per prototiping

- **LENTO** (dove il C ci mette secondi, Python ci mette minuti)
- Facile confondere tipo variabili in grossi programmi
- Parallelismo limitatissimo
- Mangia-memoria

Cython

- *Superset* di Python – ogni programma Python compila nativamente in Cython
- **COMPILATO –** il codice viene transpilato in C o C++ a seconda delle necessità ed infine compilato con GCC
- **STATIC TYPING –** si possono definire esplicitamente tutti i tipi, anche Python. Questa cosa da sola produce un boost in qualsiasi script.
- Interpoperatiblità con C (e C++)

Compilare Cython

Il modo più pratico per compilare Cython è utilizzare un *setup.py* per la costruzione del modulo in-place. In questo modo è lo script ad occuparsi autonomamente di transpilazione, compilazione e linking.

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

ext_modules = [
    Extension(
        "mandelbrot",
        ["mandelbrot.pyx"],
        extra_compile_args=['-fopenmp'],
        extra_link_args=['-fopenmp'],
    )
]

setup(
    name='parallel-mandelbrot',
    ext_modules=cythonize(ext_modules),
)
```

$ python setup.py build_ext --inplace

Documentazione

Elementi fondamentali

- cdef – definire variabili C con relativo tipo
- cpdef – definire enum/struct/funzioni C già wrappate per Python
- Python Static Typing (via cdef)
- cimport – importare moduli Cython contenenti funzioni C-compatibili

```
cdef int i, j, k
cdef float f, g[42], *h
```

```
cdef class A:
    cdef foo(self):
        print("A")

cdef class B(A):
    cdef foo(self, x=None):
        print("B", x)

cdef class C(B):
    cpdef foo(self, x=True, int k=3):
        print("C", x, k)
```

Esempi

```
from libc.stdio cimport FILE, fopen
from libc.stdlib cimport malloc, free
from cpython.exc cimport PyErr_SetFromErrnoWithFilenameObject

def open_file():
    cdef FILE* p
    p = fopen("spam.txt", "r")
    if p is NULL:
        PyErr_SetFromErrnoWithFilenameObject(OSError, "spam.txt")
    ...


def allocating_memory(number=10):
    cdef double *my_array = <double *> malloc(number * sizeof(double))
    if not my_array:  # same as 'is NULL' above
        raise MemoryError()
    ...
    free(my_array)
```

Esempi

```
from cpython.ref cimport PyObject

cdef extern from *:
    ctypedef Py_ssize_t Py_intptr_t

python_string = "foo"

cdef void* ptr = <void*>python_string
cdef Py_intptr_t adress_in_c = <Py_intptr_t>ptr
address_from_void = adress_in_c        # address_from_void is a python int

cdef PyObject* ptr2 = <PyObject*>python_string
cdef Py_intptr_t address_in_c2 = <Py_intptr_t>ptr2
address_from_PyObject = address_in_c2  # address_from_PyObject is a python int

assert address_from_void == address_from_PyObject == id(python_string)

print(<object>ptr)                     # Prints "foo"
print(<object>ptr2)                    # prints "foo"
```

Tipizzazione Statica

- Già il solo compilare un codice Python inalterato in Cython dona uno speed-up medio del **35%**
- L’utilizzo diffuso della tipizzazione statica nel semplice codice Python risulta in uno speed-up medio del **400%** rispetto al puro Python
- Lo scrivere funzioni con cpdef (funzioni in C automaticamente wrappate in Python) permette uno speed-up medio di **150x** rispetto al puro Python!

Typing Variable

```python
def f(x):
    return x ** 2 - x


def integrate_f(a, b, N):
    s = 0
    dx = (b - a) / N
    for i in range(N):
        s += f(a + i * dx)
    return s * dx

if __name__=='__main__':
    print(integrate_f(-4.,50.,10000000))
```

```python
cdef f(double x):
    return x ** 2 - x


cpdef integrate_f(double a, double b, long N):
    cdef:
        double s = 0
        double dx = (b - a) / N
    for i in range(N):
        s += f(a + i * dx)
    return s * dx
```

Runtime Pure Python - CPU times:
user 2.71 s, sys: 7 µs, total: 2.71 s

Runtime Cython - CPU times:
user 1.02 s, sys: 5.36 ms, total: 1.03 s

Runtime Cythonized Code - CPU times:
user 709 ms, sys: 9.46 ms, total: 718 ms

SpeedUp Finale: **3.82x**

Automatic Type Conversion

C types	From Python types	To Python types
[unsigned] char, [unsigned] short, int, long	int, long	int
unsigned int, unsigned long, [unsigned] long long	int, long	long
float, double, long double	int, long, float	float
char*	str/bytes	str/bytes [3]
C array	iterable	list [5]
struct, union		dict [4]

[3] The conversion is to/from str for Python 2.x, and bytes for Python 3.x.

[4] The conversion from a C union type to a Python dict will add a value for each of the union fields. Cython 0.23 and later, however, will refuse to automatically convert a union with unsafe type combinations. An example is a union of an `int` and a `char*`, in which case the pointer value may or may not be a valid pointer.

[5] Other than signed/unsigned char[]. The conversion will fail if the length of C array is not known at compile time, and when using a slice of a C array.

Cython automaticamente converte i CTypes in PyTypes e viceversa in base alla situazione ed al contesto, semplificandone molto l'utilizzo.

Supporto Nativo C++

È possibile usare il C++ come linguaggio bersaglio al posto del C, permettendoci di utilizzare funzionalità particolari quali ad esempio:

- Allocazione dinamica con keyword new e del
- Dichiarazione classi con keywork cppclass
- Supporto a funzioni e classi template
- Supporto alle overloaded functions
- Supporto all'overload di operatori C++
- extern link su puro codice C++

Attivazione C++

Di default Cython utilizza il C come linguaggio target, nonostante si possa usare pure il C++.
Per attivare il linguaggio è necessario specificare attraverso distutils mettendo dentro i file .pyx come prima riga il commento

distutils: language = c++

che dirà al setup.py di transpilare e compilare quel file come C++ anziché C.

```
# distutils: language = c++

# import dereference and increment operators
from cython.operator cimport dereference as deref, preincrement as inc

cdef extern from "<vector>" namespace "std":
```

Un semplice tutorial

Un esempio molto semplice da realizzare per iniziare è una classe in C++ puro da importare e wrappare all'interno di Cython per sveltire alcuni calcoli o il riciclaggio di codice già scritto.

Per prima cosa abbiamo una classe Rectangle e ne scriviamo header e implementazione.

Rectangle.h

```
#ifndef RECTANGLE_H
#define RECTANGLE_H

namespace shapes {
    class Rectangle {
        public:
            int x0, y0, x1, y1;
            Rectangle();
            Rectangle(int x0, int y0, int x1, int y1);
            ~Rectangle();
            int getArea();
            void getSize(int* width, int* height);
            void move(int dx, int dy);
    };
}

#endif
```

Rectangle.cpp

```cpp
#include <iostream>
#include "Rectangle.h"

namespace shapes {
    // Default constructor
    Rectangle::Rectangle () {}
    // Overloaded constructor
    Rectangle::Rectangle (int x0, int y0, int x1, int y1) {
        this->x0 = x0;
        this->y0 = y0;
        this->x1 = x1;
        this->y1 = y1;
    }
    // Destructor
    Rectangle::~Rectangle () {}
    // Return the area of the rectangle
    int Rectangle::getArea () {
        return (this->x1 - this->x0) * (this->y1 - this->y0);
    }
    // Get the size of the rectangle.
    // Put the size in the pointer args
    void Rectangle::getSize (int *width, int *height) {
        (*width) = x1 - x0;
        (*height) = y1 - y0;
    }
    // Move the rectangle by dx dy
    void Rectangle::move (int dx, int dy) {
        this->x0 += dx;
        this->y0 += dy;
        this->x1 += dx;
        this->y1 += dy;
    }
}
```

Interfaccia Cython

Ora che abbiamo il nostro codice C++ procediamo alla dichiarazione dell'interfaccia per il Cython. Le interfacce possono venir dichiarate in file separati dai file .pyx, analogamente all'accoppiata .h e .cpp. Questi file di dichiarazione sono chiamati .pyd

Rectangle.pyd

```
cdef extern from "Rectangle.cpp":
    pass

# Decalre the class with cdef
cdef extern from "Rectangle.h" namespace "shapes":
    cdef cppclass Rectangle:
        Rectangle() except +
        Rectangle(int, int, int, int) except +
        int x0, y0, x1, y1
        int getArea()
        void getSize(int* width, int* height)
        void move(int, int)
```

Si possono notare caratteristiche particolari del Cython con bersaglio C++, come ad esempio except per dichiarare se la funzione può o non può ritornare eccezioni C++.

Cython File

Andiamo ora a creare il file rect.pyx che andrà a contenere il nostro codice Cython vero e proprio di wrapping che sarà poi accessibile da un qualsiasi codice Python, anche da linea di comando.
Ma prima mostriamo un esempio di Cython Main facente uso del codice C++ e le keywork new e del.

rect.pyx (main)

```
# distutils: language = c++

from Rectangle cimport Rectangle

def main():
    rec_ptr = new Rectangle(1, 2, 3, 4)  # Instantiate a Rectangle object on the heap
    try:
        rec_area = rec_ptr.getArea()
    finally:
        del rec_ptr  # delete heap allocated object

    cdef Rectangle rec_stack  # Instantiate a Rectangle object on the stack
```

Se la classe C++ ha un costruttore di default è possibile istanziare l'oggetto anche sullo Stack.

Il Try-Finally serve a gestire eccezioni Cython riguardante la dereferenziazione di un possibile null-pointer, nel caso in cui vi sia stato un errore di allocazione e il pointer sia nullo.

Cython Wrapper Class

Il nostro obiettivo è creare una classe Cython accessibile da Python che faccia però eseguire le operazioni al sottostante codice C++.
Si procede quindi alla creazione di una classe Cython (extension type) contenente un istanza della classe C++ equivalente.

Questo extension type è accessibile da Python come una pura classe Python di tipo “built-in”, ovvero equivalente da codice a quelle integrate nel compilatore.
Tali classi hanno delle limitazioni sull’introspezione.

rect.pyx (wrapper)

```
# distutils: language = c++

from Rectangle cimport Rectangle

# Create a Cython extension type which holds a C++ instance
# as an attribute and create a bunch of forwarding methods
# Python extension type.
cdef class PyRectangle:
    cdef Rectangle c_rect  # Hold a C++ instance which we're wrapping

    def __cinit__(self, int x0, int y0, int x1, int y1):
        self.c_rect = Rectangle(x0, y0, x1, y1)

    def get_area(self):
        return self.c_rect.getArea()

    def get_size(self):
        cdef int width, height
        self.c_rect.getSize(&width, &height)
        return width, height

    def move(self, dx, dy):
        self.c_rect.move(dx, dy)
```

rect.pyx (wrapper)

```
    # Attribute access
    @property
    def x0(self):
        return self.c_rect.x0
    @x0.setter
    def x0(self, x0):
        self.c_rect.x0 = x0

    # Attribute access
    @property
    def x1(self):
        return self.c_rect.x1
    @x1.setter
    def x1(self, x1):
        self.c_rect.x1 = x1
```

```
    # Attribute access
    @property
    def y0(self):
        return self.c_rect.y0
    @y0.setter
    def y0(self, y0):
        self.c_rect.y0 = y0

    # Attribute access
    @property
    def y1(self):
        return self.c_rect.y1
    @y1.setter
    def y1(self, y1):
        self.c_rect.y1 = y1
```

Aggiungiamo inoltre delle property per l'accesso rapido agli attributi della classe bersaglio.

Costruttore Nullo(?)

Cython inizializza le classi C++ attributo di una cdef class usando il relativo costruttore nullo. Se la classe che si stà wrappando non ha tale costruttore, bisogna memorizzare un puntatore alla classe wrappata e manualmente allocare e deallocare.

Allocazione Dinamica

```
# distutils: language = c++

from Rectangle cimport Rectangle

cdef class PyRectangle:
    cdef Rectangle*c_rect  # hold a pointer to the C++ instance which we're wrapping

    def __cinit__(self, int x0, int y0, int x1, int y1):
        self.c_rect = new Rectangle(x0, y0, x1, y1)

    def __dealloc__(self):
        del self.c_rect
```

Un posto sicuro e conveniente in cui fare queste cose è all'interno dei metodi __cinit__ e __dealloc__ la cui chiamata è garantita esattamente una volta alla creazione e alla cancellazione dell'istanza Python.

Compilazione (setup.py)

```
from distutils.core import setup

from Cython.Build import cythonize

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

Se tutti i file .h, .cpp, .pyd e .pyx sono nella stessa cartella ci pensa Cython a riconoscerli e utilizzarli se richiesti. Nel caso di progetti più strutturati è comunque possibile dare posizioni, comandi di compilazione per GCC e direttive specifiche di transpiling a Cython.

Risultati

Dopo aver compilato con

`$ python setup.py build_ext –inplace`

possiamo utilizzare il modulo direttamente da un qualsiasi terminale Python:

```
>>> import rect
>>> x0, y0, x1, y1 = 1, 2, 3, 4
>>> rect_obj = rect.PyRectangle(x0, y0, x1, y1)
>>> print(dir(rect_obj))
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
 '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__',
 '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
 '__setstate__', '__sizeof__', '__str__', '__subclasshook__', 'get_area', 'get_size', 'move']
```

Template<Extra!>

```
# distutils: language = c++

# import dereference and increment operators
from cython.operator cimport dereference as deref, preincrement as inc

cdef extern from "<vector>" namespace "std":
    cdef cppclass vector[T]:
        cppclass iterator:
            T operator*()
            iterator operator++()
            bint operator==(iterator)
            bint operator!=(iterator)
        vector()
        void push_back(T&)
        T& operator[](int)
        T& at(int)
        iterator begin()
        iterator end()

cdef vector[int] *v = new vector[int]()
cdef int i
for i in range(10):
    v.push_back(i)

cdef vector[int].iterator it = v.begin()
while it != v.end():
    print(deref(it))
    inc(it)

del v
```

Cython::libcpp::extra!

```
# distutils: language = c++

from libcpp.string cimport string
from libcpp.vector cimport vector

py_bytes_object = b'The knights who say ni'
py_unicode_object = u'Those who hear them seldom live to tell the tale.'

cdef string s = py_bytes_object
print(s)  # b'The knights who say ni'

cdef string cpp_string = <string> py_unicode_object.encode('utf-8')
print(cpp_string)  # b'Those who hear them seldom live to tell the tale.'

cdef vector[int] vect = range(1, 10, 2)
print(vect)  # [1, 3, 5, 7, 9]

cdef vector[string] cpp_strings = b'It is a good shrubbery'.split()
print(cpp_strings[1])   # b'is'
```

Parallelismo!?

- CPython ha un memory model NON thread-safe, ciò ha portato l’introduzione di limitazioni affiché quel poco di parallelismo che si potesse fare fosse decente/sicuro.
- La contromisura principale è detta GIL (Global Interpreter Lock) ed è un mutex globale che avvolge l’interprete e le variabili al suo interno affinché possa essere eseguita un solo Python Bytecode per volta.

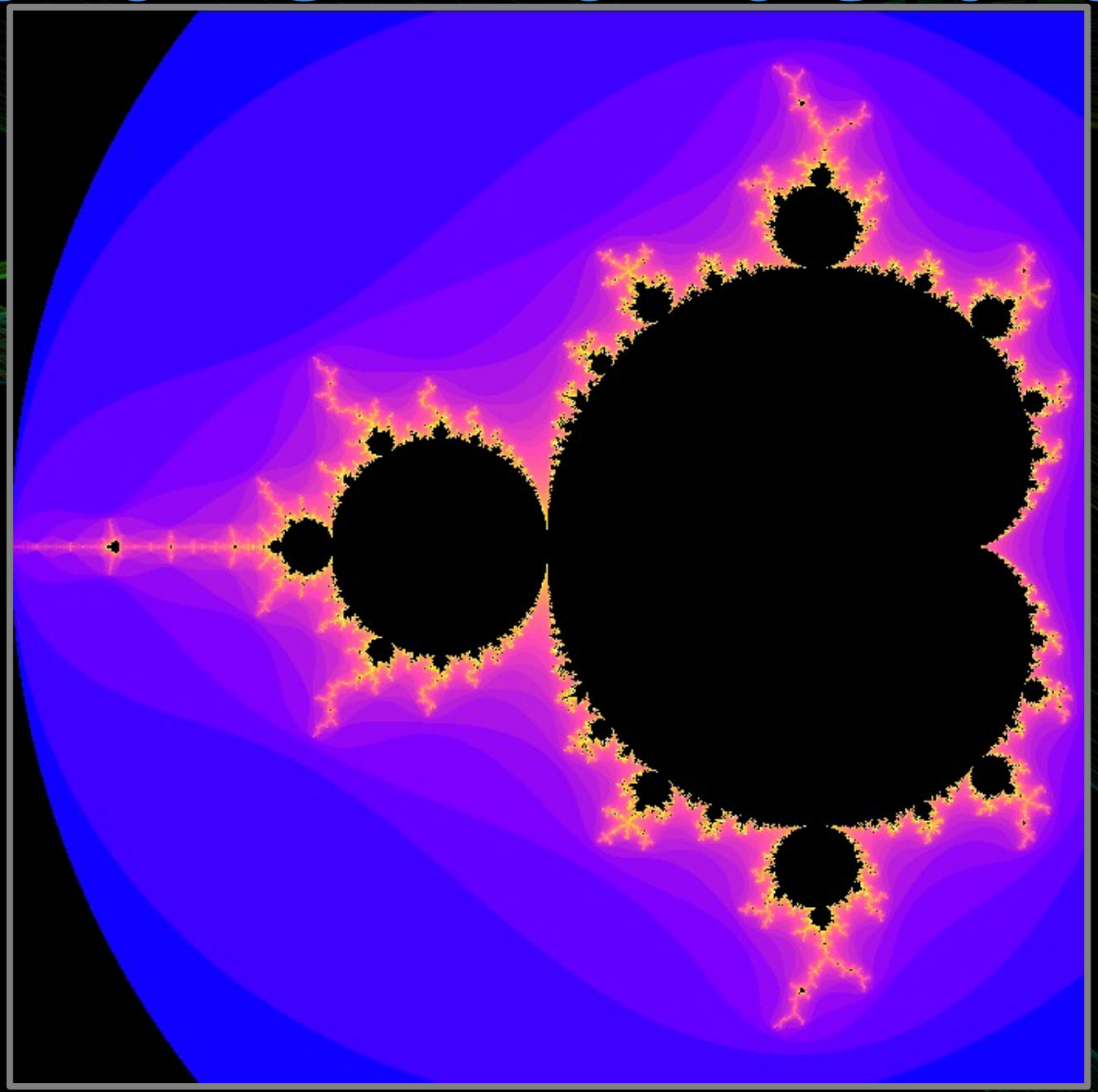
Cython & GIL

- Cython consente di disabilitare selettivamente il GIL permettendo l’esecuzione di codice puramente multithread senza le classiche limitazioni del Python. Condizione necessaria però è non accedere/modificare oggetti Python non contrassegnati come “sicuri” da modificare anche fuori dal GIL.
- La keywork da utilizzare è with nogil: e la classica indentazione.

Cython.parallel Module

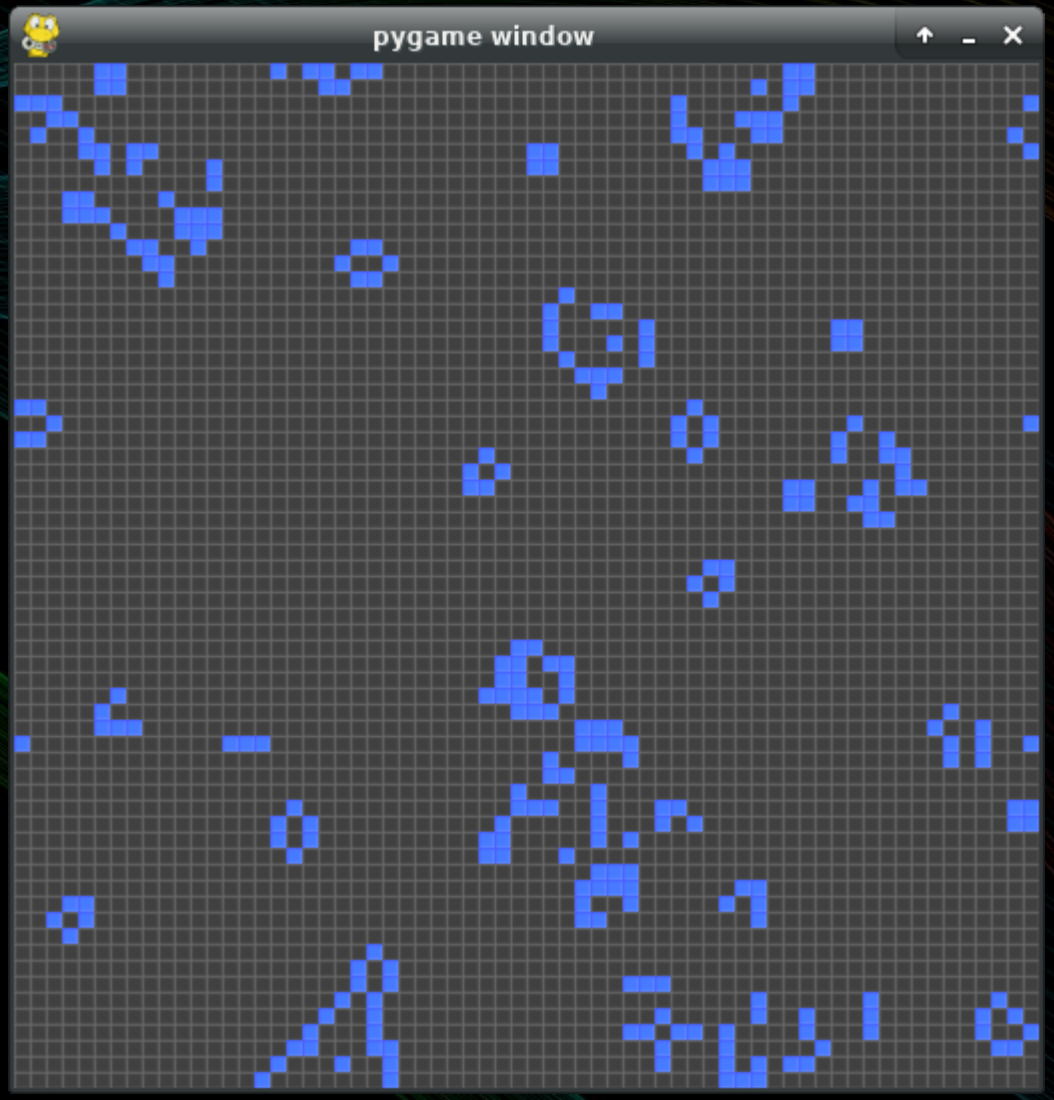
- Cython usa OpenMP per fornire il parallelismo
- In futuro vi saranno altri backend
- prange è la funzione più semplice da utilizzare e consente di lanciare thread lasciandone la gestione interamente allo scheduler di OpenMP, sceglibile via parametro
- parallel è utilizzato per l’esecuzione di uno stesso codice in più thread

Parallel Mandelbrot



Game of Life Wrapped

- Motore di calcolo in C++
- Wrapper in Cython
- Grafica in Python con PyGame
- Velocità **più che doppia** rispetto all’equivalente di puro Python, nonostante PyGame



Utilizzi nel mondo reale

- Numpy, Scipy, Scikit-learn e Pandas hanno enormi porzioni scritte in Cython, sia per l’interfaccia con codice C che per l’esecuzione veloce di bottleneck Python
- SageMath, progetto di un CAS (Computer-Algebra-System) concorrente a Mathematica per cui è stato creato l’antenato di Cython, PyRex
- Siti internet ad alto traffico usano Cython nei punti di bottleneck, ad esempio Quora

Documentazione

Cython è dotato di una fantastica User Guide in cui si trattano tutti i dettagli del linguaggio e anche altre sue caratteristiche più specifiche come l'interazione con Numpy e l'uso dei Typed MemoryView.

https://cython.readthedocs.io/en/latest/src/userguide/

Grazie dell'attenzione!

Mi trovate su Twitter come @eisterman96 , su GitLab come eisterman

Federico Pasqua

federico.pasqua.96@gmail.com