La Computación de Alto Desempeño puede ser versátil: comunicando C con Python

Pablo N Alcain, Cecilia G Jarne,
María G Molina, Rodrigo Lugones

WTPC
workshop en técnicas
de programación científica

Python

Muchísimas librerías

Sintaxis limpia

No hace falta compilar

Duck typing

Súper versátil

Python

Muchísimas librerías Sintaxis limpia No hace falta compilar

Duck typing Súper versátil

```python
# file: add_numbers.py
total = 10000000
for i in xrange(10):
  avg = 0.0
  for j in xrange(total):
    avg += j
  avg = avg/total
print "Average is
{0}".format(avg)
```

```c
/* file: add_numbers.c */
#include <stdio.h>
int main(int argc, char **argv) {
  int i, j, total;
  double avg;
  total = 10000000;
  for (i = 0; i < 10; i++) {
    avg = 0;
    for (j = 0; j < total; j++) {
      avg += j;
    }
    avg = avg/total;
  }
  printf("Average is %f\n", avg);
}
```

WTPC
workshop en técnicas de programación científica

Pero la performance, esa palabra

```
$ time python add_numbers.py
Average is 4999999.5

real 0m8.047s
user 0m7.884s
sys  0m0.012s

$ time ./add_numbers.e
Average is 4999999.500000

real 0m0.284s
user 0m0.283s
sys  0m0.001s
```

Pero la performance, esa palabra

```
$ time python add_numbers.py
Average is 4999999.5

real 0m8.047s
user 0m7.884s
sys  0m0.012s

$ time ./add_numbers.e
Average is 4999999.500000

real 0m0.284s
user 0m0.283s
sys  0m0.001s
```

28x: una simulación de 1 hora tarda 1 día!

WTPC
workshop en técnicas de programación científica

¿Qué cuenta Numpy?

```
# file: add_numbers_fast.py
from numpy import mean, arange
total = 10000000
a = arange(total)
for i in xrange(10):
  avg = mean(a)
print "Average is {0}".format(avg)
```

WTPC
workshop en técnicas de programación científica

¿Qué cuenta Numpy?

```
# file: add_numbers_fast.py
from numpy import mean, arange
total = 10000000
a = arange(total)
for i in xrange(10):
  avg = mean(a)
print "Average is {0}".format(avg)
```

```
$ time python add_numbers_fast.py
Average is 4999999.5

real 0m0.266s
user 0m0.189s
sys  0m0.075s
```

Comparable con C

WTPC
workshop en técnicas de programación científica

¿Qué cuenta Numpy?

```
# file: add_numbers_fast.py
from numpy import mean, arange
total = 10000000
a = arange(total)
for i in xrange(10):
  avg = mean(a)
print "Average is {0}".format(avg)
```

```
$ time python add_numbers_fast.py
Average is 4999999.5

real 0m0.266s
user 0m0.189s
sys  0m0.075s
```

Comparable con C

(Pero no es muy versátil, ¡sólo puedo usar vectores!)

¿Qué cuenta Numpy?

```
# file: add_numbers_fast.py
from numpy import mean, arange
total = 10000000
a = arange(total)
for i in xrange(10):
  avg = mean(a)
print "Average is {0}".format(avg)
```

```
$ time python add_numbers_fast.py
Average is 4999999.5

real 0m0.266s
user 0m0.189s
sys  0m0.075s
```

Comparable con C

(Pero no es muy versátil, ¡sólo puedo usar vectores!)

Solución manual

Si cualquier lenguaje compilado se puede linkear con cualquier otro, tiene que haber alguna forma de linkear Python con C.

1. Escribir el código en Python
2. Escribir la parte que consume tiempo en C
3. Escribir una API C/Python

¿Esto funciona? ¿La gente lo hace?

Solución manual

Si cualquier lenguaje compilado se puede linkear con cualquier otro, tiene que haber alguna forma de linkear Python con C.

1. Escribir el código en Python
2. Escribir la parte que consume tiempo en C
3. Escribir una API C/Python

¿Esto funciona? ¿La gente lo hace?

NUMPY

Herramientas

Construir el módulo de Python en C

(Si Python se hizo en C, cualquier cosa de python se puede hacer en C)

Ctypes

(Llamar a funciones de C definiendo los tipos en Python)

Cython

(Autogenerador de código de C desde Python)

F2PY

(Sólo para FORTRAN, fácil de usar, algunos problemas con memoria)

Boost, SWIG, ...

(Boost sirve mucho para C++, SWIG es para muchísimos lenguajes)

Herramientas

Construir el módulo de Python en C

(Si Python se hizo en C, cualquier cosa de python se puede hacer en C)

Ctypes

(Llamar a funciones de C definiendo los tipos en Python)

Cython

(Autogenerador de código de C desde Python)

F2PY

(Sólo para FORTRAN, fácil de usar, algunos problemas con memoria)

Boost, SWIG, ...

(Boost sirve mucho para C++, SWIG es para muchísimos lenguajes)

Librería

Partimos de una librería en C

```c
/* file: add_two.c */

float add_float(float a, float b) {
  return a + b;
}

int add_int(int a, int b) {
  return a + b;
}

int add_float_ref(float *a,
                  float *b, float *c) {
  *c = *a + *b;
  return 0;
}

int add_int_ref(int *a, int *b, int *c) {
  *c = *a + *b;
  return 0;
}
```

```c
/* file: arrays.c */

int add_int_array(int *a, int *b,
                  int *c, int n) {
  int i;
  for (i = 0; i < n; i++) {
    c[i] = a[i] + b[i];
  }
  return 0;
}

float dot_product(float *a,
                  float *b, int n) {
  float res;
  int i;
  res = 0;
  for (i = 0; i < n; i++) {
    res = res + a[i] * b[i];
  }
  return res;
}
```

WTPC
workshop en técnicas de programación científica

Librería

```
$ gcc -c -fPIC arrays.c
$ gcc -c -fPIC add_two.c
$ gcc -shared arrays.o add_two.o -o libmymath.so
$ nm -n libmymath.so
...
0000000000000730 T add_float_array
00000000000008a0 T dot_product
00000000000008d0 T add_float
00000000000008e0 T add_int
00000000000008f0 T add_float_ref
0000000000000900 T add_int_ref
...
```

Ctypes

ctypes type	C type
c_bool	_Bool
c_char	char
c_wchar	wchar_t
c_byte	char
c_ubyte	unsigned char
c_short	short
c_ushort	unsigned short
c_int	int
c_uint	unsigned int
c_long	long
c_ulong	unsigned long
c_longlong	__int64 or long long
c_ulonglong	unsigned __int64 or unsigned long long
c_float	float
c_double	double
c_longdouble	long double
c_char_p	char * (NUL terminated)
c_wchar_p	wchar_t * (NUL terminated)
c_void_p	void *

- Tipos de C
- Dynamic Loader

WTPC
workshop en técnicas de programación científica

Ctypes

Llamando a una función de la librería

```
>>> import ctypes as C
>>> math = C.CDLL('./libmymath.so')
>>> math.add_int(3, 4)
7
```

Ctypes

```
>>> math.add_float(3, 4)
```

Ctypes

```
>>> math.add_float(3, 4)
0
```

Ctypes

```
>>> math.add_float(3, 4)
0
>>> math.add_float(3.0, 4.0)
```

Ctypes

```
>>> math.add_float(3, 4)
0
>>> math.add_float(3.0, 4.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 1: <type
'exceptions.TypeError'>: Don't know how to convert
parameter 1
```

Ctypes

```
>>> math.add_float(3, 4)
0
>>> math.add_float(3.0, 4.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 1: <type
'exceptions.TypeError'>: Don't know how to convert
parameter 1
>>> math.add_float(C.c_float(3.0), C.c_float(4.0))
```

Ctypes

```
>>> math.add_float(3, 4)
0
>>> math.add_float(3.0, 4.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 1: <type
'exceptions.TypeError'>: Don't know how to convert
parameter 1
>>> math.add_float(C.c_float(3.0), C.c_float(4.0))
2
```

Ctypes: Ahora funciona

```
>>> math.add_float.restype = C.c_float
>>> math.add_float(C.c_float(3.0), C.c_float(4.0))
7.0
```

Ctypes: La forma más limpia

```
>>> math.add_float.restype = C.c_float
>>> math.add_float.argtypes = [C.c_float, C.c_float]
>>> math.add_float(3, 4)
7.0
```

Ctypes: Por referencia a int

```
>>> import ctypes as C
>>> math = C.CDLL('./libmymath.so')
>>> tres = C.c_int(3)
>>> cuatro = C.c_int(4)
>>> res = C.c_int()
>>> math.add_int_ref(C.byref(tres),
                     C.byref(cuatro),
                     C.byref(res))
0
>>> res.value
7
```

Ctypes: Por referencia a float

```
>>> import ctypes as C
>>> math = C.CDLL('./libmymath.so')
>>> tres = C.c_float(3)
>>> cuatro = C.c_float(4)
>>> res = C.c_float()
>>> math.add_float_ref(C.byref(tres),
                       C.byref(cuatro),
                       C.byref(res))
0
>>> res.value
7.0
```

¿Por qué ahora funciona?

WTPC
workshop en técnicas de programación científica

Ctypes: Arrays

A priori no debería ser muy difícil; es una llamada por referencia, apuntando al primer elemento

El problema es, como siempre, el memory management.

Manejar siempre memoria en Python

¿Cómo alocamos un array en Python?

Ctypes: Arrays

A priori no debería ser muy difícil; es una llamada por referencia, apuntando al primer elemento

El problema es, como siempre, el memory management.

Manejar siempre memoria en Python

¿Cómo alocamos un array en Python?

Ctypes: Arrays

```
>>> import ctypes as C
>>> math = C.CDLL('./libmymath.so')
>>> in1 = (C.c_int * 3) (1, 2, -5)
>>> in2 = (C.c_int * 3) (-1, 3, 3)
>>> out = (C.c_int * 3) ()
>>> math.add_int_array(C.byref(in1),
                       C.byref(in2),
                       C.byref(out),
                       3)
0
>>> out[0], out[1], out[2]
(0, 5, -2)
```

Ctypes: Numpy Arrays

```
>>> import ctypes as C
>>> import numpy as np
>>> intp = C.POINTER(C.c_int)
>>> math = C.CDLL('./libmymath.so')
>>> in1 = np.array([1, 2, -5], dtype=C.c_int)
>>> in2 = np.array([-1, 3, 3], dtype=C.c_int)
>>> out = np.zeros(3, dtype=C.c_int)
>>> math.add_int_array(in1.ctypes.data_as(intp),
                       in2.ctypes.data_as(intp),
                       out.ctypes.data_as(intp),
                       3)
0
>>> out
array([ 0,  5, -2], dtype=int32)
```

Ctypes: Structures

```c
/* file: rectangle.c */
struct _rect {
  float height, width;
};

typedef struct _rect Rectangle;

float area(Rectangle rect) {
  return rect.height * rect.width;
}
```

¿Cómo aprovechamos esta estructura como un objeto en python?

```
$ gcc -fPIC -c rectangle.c
$ gcc -shared rectangle.o -o libgeometry.so
```

Ctypes: Structures

```
# file: geometry_minimal.py
import ctypes as C
CLIB = C.CDLL('./libgeometry.so')
CLIB.area.argtypes = [C.Structure]
CLIB.area.restype = C.c_float

class Rectangle(C.Structure):
    _fields_ = [("height", C.c_float),
                ("width", C.c_float)]

def area(rect):
  return CLIB.area(rect)
```

Ctypes: Structures

```
>>> import geometry_minimal
>>> r = geometry_minimal.Rectangle()
>>> r.width = 10
>>> r.height = 30
>>> geometry_minimal.area(r)
300.0
```

Ctypes: Structures

```
# file: geometry.py
import ctypes as C
CLIB = C.CDLL('./libgeometry.so')
CLIB.area.argtypes = [C.Structure]
CLIB.area.restype = C.c_float

class Rectangle(C.Structure):
    _fields_ = [("height", C.c_float),
                ("width", C.c_float)]

    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return CLIB.area(self)
```

Ctypes: Structures

```
# file: geometry.py
import ctypes as C
CLIB = C.CDLL('./libgeometry.so')
CLIB.area.argtypes = [C.Structure]
CLIB.area.restype = C.c_float

class Rectangle(C.Structure):
    _fields_ = [("height", C.c_float),
                ("width", C.c_float)]

    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return CLIB.area(self)
```

Llamar a las funciones

Ctypes: Structures

```
# file: geometry.py
import ctypes as C
CLIB = C.CDLL('./libgeometry.so')
CLIB.area.argtypes = [C.Structure]
CLIB.area.restype = C.c_float

class Rectangle(C.Structure):
    _fields_ = [("height", C.c_float),
                ("width", C.c_float)]

    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return CLIB.area(self)
```

Llamar a las funciones

Hereda de C struct

Ctypes: Structures

```
# file: geometry.py
import ctypes as C
CLIB = C.CDLL('./libgeometry.so')
CLIB.area.argtypes = [C.Structure]
CLIB.area.restype = C.c_float

class Rectangle(C.Structure):
    _fields_ = [("height", C.c_float),
                ("width", C.c_float)]

    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return CLIB.area(self)
```

Llamar a las funciones

Hereda de C struct

El mismo memory layout

Ctypes: Structures

```
# file: geometry.py
import ctypes as C
CLIB = C.CDLL('./libgeometry.so')
CLIB.area.argtypes = [C.Structure]
CLIB.area.restype = C.c_float

class Rectangle(C.Structure):
    _fields_ = [("height", C.c_float),
                ("width", C.c_float)]

    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return CLIB.area(self)
```

Llamar a las funciones

Hereda de C struct

El mismo memory layout

Constructor

WTPC
workshop en técnicas de programación científica

Ctypes: Structures

```
# file: geometry.py
import ctypes as C
CLIB = C.CDLL('./libgeometry.so')
CLIB.area.argtypes = [C.Structure]
CLIB.area.restype = C.c_float

class Rectangle(C.Structure):
    _fields_ = [("height", C.c_float),
                ("width", C.c_float)]

    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return CLIB.area(self)
```

Llamar a las funciones

Hereda de C struct

El mismo memory layout

Constructor

Wrapper a C

Ctypes: Structures

```
>>> import geometry
>>> r = geometry.Rectangle(2, 3)
>>> r.area()
6.0
>>> r.width=10
>>> r.area()
30.0
```

La implementación en C está completamente encapsulada

La Computación de Alto Desempeño puede ser versátil: comunicando C con Python

Pablo N Alcain, Cecilia G Jarne,
María G Molina, Rodrigo Lugones

WTPC
workshop en técnicas
de programación científica