Universidade Federal do Paraná

Fernando Rodrigo Bilinski

Otimização de código através do uso de interfaces entre Python e C

Curitiba PR
2017

Fernando Rodrigo Bilinski

Otimização de código através do uso de interfaces entre Python e C

Trabalho apresentado como requisito parcial à conclusão do Curso de Bacharelado em Ciência da Computação, setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Prof. Dr. Renato Carmo.

Curitiba PR
2017

Termo de aprovação

Esta folha deve ser substituída pela ata de defesa ou termo de aprovação devidamente assinado, que será fornecido pela secretaria do programa após a defesa ter sido concluída e aprovada (vide arquivo `aprovacao.tex`*).*

Aos nossos professores, pelo carinho,
dedicação e esforço em lecionar.

Agradecimentos

A minha família, em especial a minha mãe.

Ao meu orientador, Prof. Renato, por acreditar nesse trabalho e por todo seu esforço.

A todos meus professores por terem contribuído para minha formação.

Aos meus amigos que nos momentos difíceis me motivaram a seguir em frente.

Resumo

Em sua dissertacão, [Züge, 2012] descreveu um algoritmo genérico de Branch & Bound para solução exata do Problema da Clique Máxima, apresentando uma implementação em linguagem Python. Como esta é uma linguagem interpretada, ela não é executada diretamente pelo sistema operacional, tendo assim desempenho reduzido em comparação a linguagens compiladas. Neste trabalho são apresentadas três ferramentas que permitem ao código Python acesso a bibliotecas escritas em C, com exemplificação de uso, permitindo assim a implementação de rotinas críticas da implementação de [Züge, 2012] na linguagem C, contornando assim o problema de desempenho do Python.

Palavras-chave: otimização, clique máxima, python, cython, swig, C.

Abstract

In his dissertation, [Züge, 2012] described a general Branch & Bound algorithm for the exact solution of the Maximum Clique Problem, presenting a Python language implementation. Python is an interpreted language that is not executed directly by the operating system, thus reducing performance compared to compiled languages. We discuss and give examples on three tools that provide access to C compiled libraries from Python code, allowing the implementation of the critical routines of [Züge, 2012] in the C language, avoiding the Python performance problem.

Keywords: optimization, maximum clique, python, cython, swig, C.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Acrônimos

SWIG Simplified Wrapper and Interface Generator

Lista de Símbolos

Capítulo 1

Introdução

1.1 Desafio

Otimizar os algoritmos implementados em Python para encontrar a solução exata do problema da clique máxima escritos por [Züge, 2012].

Sendo Python uma linguagem interpretada, o código é executado por um interpretador e não diretamente pelo sistema operacional. Isso permite a linguagem ser portável a qualquer plataforma no qual o seu interpretador esteja disponível, sem necessidade de alteração do código, porém isto também traz como desvantagem o fato de que o processo de interpretação tem um custo computacional, deixando o tempo de execução do código maior do que um similar escrito em uma linguagem compilada, como por exemplo a linguagem C.

1.2 Motivação

Contornar o problema de desempenho do Python ao conseguir executar rotinas complexas em menor tempo.

1.3 Proposta

Implementar em C as rotinas críticas dos algoritmos e utilizá-las no código Python através de interfaces entre Python e C. Foi usada como candidata a ser implementada em C a função *intersect* da solução de [Züge, 2012]. Esta função devolve uma lista contendo a interseção entre duas listas.

1.4 Contribuição

Fornecer um guia sobre como criar bibliotecas escritas em C e utilizá-las no Python.

1.5 Organização do documento

Este trabalho apresentará três soluções disponíveis para fazer interface entre Python e C e testes realizados com elas.

1. O capítulo 2 trata o SWIG, uma ferramenta especializada em criar interfaces entre bibliotecas escritas C/C++ e diversas linguagens.

2. O capítulo 5 descreverá os testes realizados com o SWIG usando funções matemáticas escritas em Python.

3. O capítulo 3 mostra o Cython, um compilador estático que pode converter código escrito em Python para a linguagem C ou C++.

4. O capítulo 6 descreverá os testes realizados com o Cython usando funções matemáticas escritas em Python.

5. O capítulo 7 descreverá os testes realizados com o Cython usando a função *intersect* da solução de [Züge, 2012].

6. O capítulo 4 apresentará o módulo ctypes da linguagem Python, que permite carregar bibliotecas compartilhadas escritas na linguagem C ou C++ e utilizá-las como se fossem módulos do Python.

7. O capítulo 8 descreverá os testes ao utilizar o módulo ctypes do Python com funções matemáticas escritas em Python.

Capítulo 2

SWIG

O SWIG é uma ferramenta de desenvolvimento de software que permite conectar programas escritos em C ou C++ com programas escritos em outras linguagens de programação através do uso de interfaces. Ele automatiza em grande parte essa tarefa de integração gerando os códigos-fontes das interfaces baseando-se em declarações de variáveis e protótipos de funções.

Ele é frequentemente utilizado para criar ambientes de programação interpretados ou compilados para programas C ou C++.

Até sua versão 3.0, o SWIG possui suporte para gerar interfaces para as seguintes linguagens: Javascript, Perl, PHP, Python, Tcl, Ruby, C#, Common Lisp, D, Go language, Java, Lua, Modula-3, OCAML, Octave, Scilab, R, Guile, MzScheme/Racket e Chicken.

2.1 Origem

Originalmente desenvolvido em 1995, o SWIG foi usado inicialmente no departamento de física teórica do Laboratório Nacional de Los Alamos para construir interfaces de usuário para códigos de simulação. Os cientistas precisavam trabalhar com grandes quantidades de dados de simulação e constantes mudanças no código base. O uso de interfaces usando linguagens interpretadas era uma solução simples mas ainda assim altamente flexível para resolver esses problemas.

Apesar de ter sido originalmente desenvolvido para aplicações científicas, o SWIG evoluiu para uma ferramenta de propósito geral usada em uma vasta variedade de aplicações.

2.2 Exemplo de uso

Considere o seguinte código em C.

```
/* Arquivo: exemplo.c */

double a = 3.0;

/* Calcula o fatorial de n */
int fact(int n) {
  if (n <= 1)
    return 1;
  else
    return n*fact(n-1);
}

/* Calcula o módulo de n por m */
int modulo(int n, int m) {
  return(n % m);
}
```

Para poder acessar essas funções e a variável a partir de uma outra linguagem, é preciso criar um arquivo de interface que dirá ao SWIG o que deverá ser disponibilizado. Por convenção arquivos de interface possuem extensão **.i**.

Em sua estrutura básica o arquivo de interface deverá conter:

1. A diretiva **%module**, que definirá o nome do módulo;
2. As declarações das variáveis e protótipos das funções que serão acessíveis pela interface.
3. O bloco **%{ %}**, que permite adicionar código extra, como *include* de arquivos *header* ou declarações adicionais de funções e variáveis que serão inseridos posteriormente no código gerado;

Abaixo tem-se um exemplo de um arquivo de interface para o arquivo **exemplo.c**.

```
/* File : exemplo.i */

%module exemplo
%{
  extern double a;
  extern int fact(int);
  extern int modulo(int n, int m);
%}

extern double a;
extern int fact(int);
extern int modulo(int n, int m);
```

Tendo o arquivo **.c** com as variáveis e funções a serem acessadas e o arquivo **.i** que dirá como será a interface, pode-se então gerar o módulo *wrapper*[1] para a linguagem desejada.

No exemplo abaixo, o módulo será gerado para a linguagem Python.

[1]Um wrapper consiste de um código em que o principal propósito é executar um segundo código. É usado geralmente para simplificar interfaces complicadas ou para permitir que códigos incompatíveis possam trabalhar juntos.

```
$ swig -python exemplo.i
$ gcc -c -fpic exemplo.c exemplo_wrap.c -I/usr/local/include/python2.7
$ gcc -shared exemplo.o exemplo_wrap.o -o _exemplo.so
```

No primeiro passo, usando o arquivo de interface **exemplo.i**, gerou-se o arquivo **exemplo.py**, que serve para abstrair o acesso ao módulo no Python, e o arquivo de wrapper **exemplo_wrap.c**, que é responsável por tornar possível o acesso externo de nosso código no arquivo **exemplo.c**.

No segundo passo compilou-se os arquivos **exemplo.c** e **exemplo_wrap.c** para gerar arquivos objetos. A opção **-fPIC** *(Position Independent Code)* é necessária para que a biblioteca use endereços relativos de memória.

No terceiro passo gerou-se a biblioteca compartilhada com os arquivos objetos gerados no segundo passo. A opção **-shared** serviu para criar um *Shared Object*, o que permite ligar a biblioteca gerada com outros objetos para formar um arquivo executável.

Após esses passos o módulo escrito em C já está disponível para ser usado no Python.

```
1 import exemplo
2
3 exemplo.fact(4)
4 exemplo.modulo(23, 7)
5 exemplo.cvar.a + 4.5
```

2.3 Conversões automáticas

O SWIG pode realizar conversões automáticas de tipos de dados entre a linguagem C e outras linguagens.

Tabela 2.1: Tabela de conversão de tipos Python/C no SWIG

Python	C
int	int
float	double
string	char *
complex	—

O tipo **float** do Python é equivalente ao **double** do C, por isso o ideal é utilizar o double como tipo de ponto flutuante nos códigos C se estes forem utilizados pelo código Python.

O tipo complex do Python não tem uma equivalência direta com nenhum tipo do C, consta na documentação do Python é que ele é armazenado como uma struct. [Python Software Foundation, 2017a]

```
1 typedef struct {
2   double real;
3   double imag;
4 } Py_complex;
```

2.4 Typemaps

As linguagens de programação se distinguem entre si por diversas características. Uma delas é o conjunto de seus tipos de dados. Enquanto para algumas o tipo de dado *string*, por

exemplo, é um tipo primitivo, na linguagem C *string* é um tipo de dado composto, formado por um conjunto de dados do tipo *char*.

Sendo assim, ao construir-se uma interface no SWIG, pode-se acabar esbarrando com a dificuldade de trafegar dados entre duas linguagens de programação onde não há uma conversão automática, nesses casos, o SWIG oferece como opção os **typemaps**.

Typemaps permitem definir uma conversão de tipos ou estruturas de dados de uma linguagem para uma representação da mesma em C ou C++ e vice-versa.

2.4.1 Formato da declaração

Uma declaração *typemap* possui a seguinte forma.

```
%typemap(method [, modifiers]) typelist code ;
```

- *method* especifica o tipo do *typemap*. Os valores disponíveis são *in*, *out*, *argout*.
- *modifiers* é uma lista opcional de valores `name="value"` separados por vírgula. São usados para adicionar informação extra ao *typemap* e muitas vezes são dependentes da linguagem de destino.
- *typelist* é uma lista de padrões de tipos de dados da linguagem C a serem encontrados.
- *code* especifica o código que será utilizado pelo *typemap* para realizar a conversão.

2.4.2 Variáveis especiais

Para identificar genericamente as variáveis que o *typemap* irá usar nas conversões, pode-se usar as seguintes variáveis especiais.

$input: Representa a variável da linguagem destino que está sendo convertida para o C.

$result: Representa a variável que será retornada do C para a linguagem de destino.

$1 ... $n: Representa as variáveis da linguagem C declaradas no *typemap*. No caso de múltiplos argumentos $1 representa a primeira variável, $2 a segunda e assim por diante.

Antes de ser gerado o código de conversão de tipo, as variáveis especiais são substituídas pelos seus valores reais pelo SWIG.

2.4.3 Exemplos de uso

Pode-se mapear a conversão de um tipo de dado.

```
%typemap(in) int{}
```

Pode-se mapear vários tipos.

```
%typemap(in) int, short, long int {}
```

Também é possível mapear por nome de variável.

```
%typemap(in) int size {}
```

Ou ainda múltiplos argumentos [SWIG, 2017].

```
%typemap(in) (int argc, char *argv[]){}
```

O SWIG insere o código de conversão de tipo do *typemap* dentro do código de cada arquivo gerado com sufixo **_wrap.c** em que o padrão case.

Por exemplo, adicionando o seguinte typemap no arquivo de interface ao Python.

```
/* Conversão de Python para C */
%typemap(in) int {
  $1 = PyInt_AsLong($input);
}

/* Conversão de C para Python */
%typemap(out) int {
  $result = PyInt_FromLong($1);
}
```

E tendo a seguinte função na linguagem C.

```
int fatorial(int x);
```

O código da função de interface gerada será algo como:

```
PyObject *wrap_fatorial(PyObject *self, PyObject *args) {
  int arg1;
  int result;
  PyObject *obj1;
  PyObject *resultobj;

  if (!PyArg_ParseTuple("OO:fatorial", &obj1)) return NULL;

  /* typemap "in", argumento 1 */
  {
    arg1 = PyInt_AsLong(obj1);
  }

  result = fatorial(arg1);

  /* typemap "out", retornando o valor */
  {
    resultobj = PyInt_FromLong(result);
  }

  return resultobj;
}
```

Pode-se observar a conversão do parâmetro, que é um objeto Python, para o tipo *int* antes da chamada da função *fatorial* e o tratamento do resultado antes de retorna-lo ao Python.

2.5 Conclusão

O SWIG é uma ferramenta para gerar código de interface baseado em arquivos que contém as definições do que deverá estar disponível no módulo de interface e como lidar com as conversões de tipos de dados que não são automáticas. Como resultado final temos o módulo *wrapper* e o código para importar o módulo na linguagem destino. Diante da complexidade para

se criar um módulo em C para qualquer das linguagens suportadas pelo SWIG, o esforço do usuário com o auxílio desta ferramenta é bem reduzido.

Porém o SWIG não altera código do usuário, nem converte módulos escritos em outras linguagens para a linguagem C. Esse trabalho, ao menos para a linguagem Python, pode ser realizado com o uso de uma ferramenta chamada Cython.

Capítulo 3

Cython

O Cython é um compilador estático otimizador que pode compilar código da linguagem Python e produzir uma versão em C do mesmo, que depois pode ser compilada como uma biblioteca compartilhada e pode ser importado como um módulo por qualquer script Python.

Ele também é uma linguagem de programação, que é um superconjunto da linguagem Python, adicionado a esta suporte a chamada de funções em C e declaração estática de tipos de dados do C para variáveis e atributos de classes do Python. Essas funcionalidades da linguagem permitem ao desenvolvedor indicar ao compilador Cython que um objeto Python que representa um tipo de dado pode ser substituído por um tipo estático do C, otimizando e reduzindo assim o código gerado.

Não se deve confundi-lo com o CPython, que é a implementação oficial da linguagem Python escrita em linguagem C [Python Software Foundation, 2017d].

3.1 Origem

O projeto Cython baseou-se em outro projeto chamado Pyrex [Ewing, 2017], e seu desenvolvimento foi parcialmente motivado por necessidades do SageMath [SageMath, 2017], um software matemático-científico open-source que utiliza o Cython para acelerar seus algoritmos.

3.2 Compilação do código

Diferente do Python que é uma linguagem interpretada, para poder utilizar o Cython deve-se compilar o código. O compilador Cython pode compilar a linguagem Python, já que esta é englobada pela linguagem Cython. Não é necessário fazer modificações no código, com exceção de algumas funcionalidades ainda não suportadas pelo compilador.

É uma prática comum usar a extensão **.pyx** para identificar os arquivos que serão compilados.

O processo de compilação acontece em duas etapas:

1. O arquivo **.pyx** é compilado pelo Cython e um arquivo **.c** é gerado.
2. O arquivo **.c** é compilado pelo compilador C em um arquivo **.so** no Linux ou **.pyd** no Windows, gerando uma biblioteca compartilhada que pode ser utilizada pelo Python.

Essas duas etapas podem ser executadas de várias maneiras:

- Com um arquivo **setup.py** que utilize o pacote *distutils* do Python.

- Usa-se o comando *pyximport* dentro do código Python para importar os arquivos **.pyx**, o que acabará utilizando o *distutils* em segundo plano.
- Chama-se o compilador Cython pela linha de comando para gerar o arquivo **.c** e depois usa-se o compilador C para gerar a biblioteca compartilhada.
- Caso esteja-se usando o software Jupyter [Project Jupyter, 2017] ou o SageMath [SageMath, 2017] para desenvolver, basta usar o código inline.

Dentre as opções de compilação, a com o auxílio do *distutils* é a mais usada.

3.2.1 Cython pela linha de comando

Exemplo de Makefile para gerar uma biblioteca compartilhada com o Cython.

```
PYTHON_INCLUDE = /usr/include/python$(PYTHON_VERSION)

# Compila o arquivo Python com o Cython
libteste.c: libteste.pyx
	cython libteste.pyx

# Compila o arquivo C gerado pelo Cython
libteste.o: libteste.c
	$(CC) -c -fPIC -I S(PYTHON_INCLUDE) libteste.c

# Liga o arquivo objeto em uma biblioteca compartilhada
libteste.so: libteste.o
	$(CC) -shared libteste.o -o libteste.so
```

A opção **-fPIC** *(Position Independent Code)* é necessária para que a biblioteca use endereços relativos de memória. Já a opção **-shared** serve para criar um *Shared Object*, o que permite ligar a biblioteca gerada com outros objetos para formar um executável.

Após isso já é possível importar a biblioteca pelo código Python.

```
import libteste
```

3.2.2 Compilação do código com o distutils e o cythonize

O *distutils* [Python Software Foundation, 2017c] é um pacote Python que auxilia na compilação e instalação de módulos adicionais no Python.

Para utiliza-lo, primeiro deve-se criar um arquivo chamado **setup.py** como no modelo abaixo.

```
from distutils.core import setup
from Cython.Build import cythonize
setup(
  name = "Intersect function",
  ext_modules = cythonize('intersect.pyx'),
)
```

Importa-se a função *setup* do *distutils* e a *cythonize* do Cython.

A função *cythonize* irá compilar os arquivos para C e retornar uma lista de extensões para a função *setup* do *distutils*. É possível usar caracteres curinga para informar os arquivos a serem compilados.

Já a função *setup* irá gerar as bibliotecas compartilhadas. Ela é responsável por definir os parâmetros de compilação, verificar caminhos dos arquivos *header*, compilar os arquivos, definir dados sobre o módulo, instalá-los no Python e tudo isso com apenas poucas linhas.

É possível passar parâmetros em linha de comando ao executar o arquivo **setup.py** para definir qual ação ele deve tomar, como executar um comando como **build**, **install**, **clean** entre outros ou definir parâmetros de compilação como ativar o debug do **gdb**, indicar outros caminhos para bibliotecas ou arquivos includes ou simplesmente verificar informações do módulo.

Com o arquivo **setup.py** pronto, basta chamá-lo com os parâmetros desejados.

```
$ python setup.py build_ext --inplace
Compiling intersect.pyx because it changed.
[1/1] Cythonizing intersect.pyx
running build_ext
building 'intersect' extension
creating build
creating build/temp.linux-x86_64-2.7
x86_64-linux-gnu-gcc -pthread -DNDEBUG -g -fwrapv -O2 -Wall
   -Wstrict-prototypes -fno-strict-aliasing -Wdate-time -D_FORTIFY_SOURCE=2
   -g -fstack-protector-strong -Wformat -Werror=format-security -fPIC
   -I/usr/include/python2.7 -c intersect.c
   -o build/temp.linux-x86_64-2.7/intersect.o
x86_64-linux-gnu-gcc -pthread -shared -Wl,-O1 -Wl,-Bsymbolic-functions -Wl,
   -Bsymbolic-functions -Wl,-z,relro -fno-strict-aliasing -DNDEBUG -g
   -fwrapv -O2 -Wall -Wstrict-prototypes -Wdate-time -D_FORTIFY_SOURCE=2
   -g -fstack-protector-strong -Wformat -Werror=format-security -Wl,
   -Bsymbolic-functions -Wl,-z,relro -Wdate-time -D_FORTIFY_SOURCE=2 -g
   -fstack-protector-strong -Wformat
   -Werror=format-security build/temp.linux-x86_64-2.7/intersect.o -o
   /home/fernando/UFPR/TG/fb/src/cython/Intersect/teste-2/intersect.so
```

Neste exemplo usamos o comando *build_ext* para que seja criado um módulo Python escrito em C/C++. Como parâmetro para o comando *build_ext* usou-se **–inplace** para que ele ignore o diretório padrão de módulos compilados e deixe os arquivos no diretório atual.

3.3 Otimização do código

Mesmo que se esteja compilando código Python puro já se pode perceber um aumento no desempenho do mesmo. Para ter o máximo de ganho, o ideal é realizar modificações no código usando a linguagem *Cython* para auxiliar o compilador nas otimizações.

As modificações se baseiam em declarações de **tipagem estática** de variáveis e parâmetros, onde indicamos ao compilador que ele pode ignorar a tipagem dinâmica do Python, onde os tipos de dados são representados por objetos, e usar os tipos estáticos do C, deixando o código gerado mais simples e rápido.

Entretanto, as declarações de tipo devem ser usadas só quando há um bom motivo para acreditar que elas farão diferença, pois elas podem deixar o código menos legível.

Pode haver casos em que seja necessário o uso dos tipos dinâmicos do Python e nesse ponto o Cython pode falhar e acabar realizando conversões de tipo desnecessárias, o que deixará o código pior do que antes. Por isso é preciso fazer a **tipagem estática** apenas nos lugares certos.

3.3.1 Variáveis e parâmetros

Variáveis no *Cython* é simples para as conversões automáticas, basta adicionar a palavra chave **cdef** e o tipo do dado conforme sintaxe da linguagem C. No caso de parâmetros de funções não há necessidade de usar o **cdef**.

Tome como base o seguinte código.

```
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
```

Após as tipagens ele ficará deste modo.

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

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

Com a variável de iteração **i** sendo do tipo inteiro do C, o laço **for** será compilado em código C puro. As variáveis **a**, **s**, **x** e **dx** participam de operações aritméticas dentro do laço, portanto a tipagem delas também é importante.

Tabela 3.1: Conversões automáticas de tipos entre Python e C no Cython

Tipos de dados do C	**De tipos do Python**	**Para tipos do Python**
[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 [a]
struct, union		dict [b]

[a] A conversão é de/para *str* no Python 2 e *bytes* no Python 3.

[b] A conversão de um tipo *union* do C para o tipo *dict* do Python irá adicionar um valor para casa um dos campos da *union*. O Cython 0.23 e posteriores, podem recusar de fazer a conversão automática da *union* para uma combinação de tipos não-segura, como *int* e *char**, já que *char** pode ser um ponteiro válido ou não.

3.3.2 Funções

As chamadas de função do Python podem ser dispendiosas e no Cython elas podem ficar piores já que pode ser necessário realizar conversões para poder fazer a chamada, além de

que para a função ser acessível da área do python é necessário que o Cython adicione **funções wrappers** [1] o que diminui o desempenho e aumenta o código C gerado.

No Cython funções podem ser definidas de três formas:

1. **def**: Chamável apenas dentro de código Python.
2. **cdef**: Chamável apenas dentro de código C.
3. **cpdef**: Chamável tanto do Python quanto do C, pois apesar de a função ser definida dentro do C ainda é criado o **wrapper** para chamadas do Python, contudo se a função for chamada dentro de código Cython usa as convenções de chamada do C para ser mais rápido.

3.4 Conclusão

O Cython é uma ferramenta de uso simples mas que gera um grande impacto no desempenho de programas ao reescrever seus códigos da linguagem interpretada Python em código da linguagem C o qual é executado diretamente pelo sistema operacional, dando assim ao desenvolvedor as vantagens do Python com o poder da linguagem C.

As alterações necessárias no código Python para se obter otimização máxima são pequenas, mas demandam alguma análise, se for concentrado esforço apenas nas porções mais críticas do programa, em módulos ou funções mais utilizados, pode-se assim reduzir a refatoração de código e manter o ganho de desempenho.

Ele também pode usar código escrito originalmente em C/C++, inclusive bibliotecas compiladas do sistema, funcionalidade esta semelhante ao do módulo *ctypes* do Python [Behnel et al., 2017a].

[1] Funções wrappers são funções encapsuladoras que criam uma camada de acesso para uma função ou variável que não está disponível. No caso do Cython, os wrappers são usados para comunicar entre as áreas de trabalho do Python e do C.

Capítulo 4

Ctypes

O ctypes é um módulo nativo da linguagem Python que permite a chamada de funções escritas em C/C++ e armazenadas em dll's ou bibliotecas compartilhadas, além de permitir criar, acessar e manipular tipos de dados do C no Python [Python Software Foundation, 2017b].

4.1 Compilação de código C para gerar uma biblioteca

Para poder ser importada pelo ctypes a biblioteca precisa ser compilada como biblioteca compartilhada [(PGI/JCNS-TA), 2017].

```
1 # Compila em uma biblioteca compartilhada
2 integral.so: integral.c
3       $(CC) -fPIC -shared -o integral.so integral.c
```

A opção **-fPIC** *(Position Independent Code)* é necessária para que a biblioteca use endereços relativos de memória. Já a opção **-shared** serve para criar um *Shared Object*, o que permite ligar a biblioteca gerada com outros objetos para formar um executável.

Não há nenhuma particularidade na criação da biblioteca para o módulo ctypes, tanto que a mesma biblioteca pode também ser usada por programas escritos em C.

4.2 Uso da biblioteca dentro do código Python

```
1 # Importa ctypes
2 from ctypes import *
3
4 # Importa a biblioteca usando o ctype
5 lib = CDLL( "./integral.so");
```

Após ter-se importado o módulo ctypes, usa-se a função *CDLL* no Linux (ou *Windll* no caso do Windows) para carregar a biblioteca compartilhada. Dependendo da configuração do sistema é necessário adicionar './' na frente do nome da biblioteca para indicar que o arquivo deve ser buscado na pasta atual. No Windows não é necessário colocar a extensão do arquivo, ao contrário do Linux.

A função devolve um apontador para o acesso à biblioteca. As funções da biblioteca carregada podem ser acessadas como membros de uma classe por meio desse apontador, por exemplo, como *lib.integrate_f()*.

Toda função importada é considerada sem parâmetros e com tipo do valor de retorno *int*. O módulo ctypes não sabe quais são os tipos de dados, então após o carregamento da biblioteca é necessário informar a assinatura da função.

```
1 # Informando o tipo de valor de retorno da função e argumentos
2 lib.integrate_f.restype = c_double
3 lib.integrate_f.argtypes = ( c_double, c_double, c_int )
```

No exemplo acima foi alterado o tipo do retorno da função para *double* no atributo *restype* e, foram adicionados três parâmetros para a função com seus tipos, no atributo *argtypes*.

Após esse passo a biblioteca externa já está disponível para uso no código Python.

4.3 Conclusão

A função do módulo **ctypes** do Python é semelhante ao SWIG e ao Cython na parte que se trata de permitir acesso a bibliotecas escritas em C pelo Python. Porém o ctypes não possui os typemaps do SWIG que permitem a conversão de tipos de dados diferentes entre as linguagens e nem a capacidade de converter código Python em C do Cython.

A única vantagem identificada é que ele é um módulo nativo da linguagem Python e portanto não demanda instalação. Dependendo de qual uso se queira fazer e se não estiverem envolvidas estruturas de dados complexas, ele pode ser uma opção atraente.

Capítulo 5

Testes com o SWIG acessando bibliotecas escritas em C

Testou-se a integração entre programas de linguagens diferentes através do uso de interfaces do SWIG com a chamada de uma função de uma biblioteca escrita em C a partir de um programa escrito em Python.

Para a realização dos testes foi utilizado notebook com a seguinte configuração: Processador Intel Core2 Duo T6600, com dois núcleos de 2.20 GHz, 4 GB de memória RAM, sistema operacional Linux 64 bits, gcc versão 5.4.0

5.1 Sequência dos testes e resultados

5.1.1 Chamadas de função com parâmetros

Foi testada a passagem de dados do Python por parâmetros para funções escritas em linguagem C.

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int fact(int n){
5        if (n <= 1){
6              return 1;
7        }else{
8              return n*fact(n-1);
9        }
10 }
11
12 void imprime( char *texto ){
13       printf("%s\n", texto );
14 }
15
16 double potencia( double base, double potencia ){
17       return pow( base, potencia );
18 }
```

- Na função *fact* testou-se a passagem e retorno de tipo int.
- Na função *imprime* testou-se passagem de tipo string sem retorno.

- Na função *potencia* testou-se a passagem de dois parâmetros float e o retorno de um double.

Criou-se o seguinte arquivo de interface para as funções C acima.

```
/*
  Arquivo funcoes.i
  Interface SWIG: C <-> Python
*/
%module funcoes
%{
      int fact(int n);
      void imprime( char *texto );
      double potencia( double base, double potencia );
%}

int fact(int n);
void imprime( char *texto );
double potencia( double base, double potencia );
```

Conclusão do teste: As três funções retornaram os resultados esperados para as entradas dadas. As conversões de tipos descritas na seção 2.3 ocorreram como esperado.

5.1.2 Uso de listas e dicionários Python como parâmetros

Não existem conversões automáticas dos tipos lista e dicionários do Python para qualquer estrutura de dados na linguagem C. Portanto fez-se necessário o uso de *typemaps* para converter o tipo lista do Python em um par de parâmetros no C.

Para armazenar as listas e dicionários do Python criou-se uma lista ligada.

```
typedef struct lista_s *lista;
typedef struct no_lista_s *no_lista;

// Cabeça da lista
struct lista_s{
      size_t tamanho;
      no_lista *elemento;
};

// Nó da lista
struct no_lista_s{
      int tipo;
      void *valor;
};
```

Tanta a lista quanto o dicionário do Python permitem que sejam armazenados elementos de tipos distintos.

- **Lista**

```
lista = [1, 2, 4.2, 'string']
```

- **Dicionário**

```
dic = {1 : 'inteiro', 4.2 : 'double', 2 : 42, 3 : 7.2, 'a' : 'Python'}
```

Por esse motivo, definiu-se o campo **valor** da *struct* no_lista_s como sendo void*, o que permite armazene-se vários tipos de dados. Porém o tipo (void *) utiliza 4 bytes, o que é suficiente para armazenar int ou ponteiros mas o tipo double usa 8 bytes, então no teste atribui-se um ponteiro ao double e o valor do ponteiro colocado no campo tipo (void *).

Conclusão do teste: As funções testadas executaram conforme o esperado, mesmo as listas contendo vários tipos de dados.

Durante a implementação dos *typemaps* fez-se o uso de funções de manipulação de dados do próprio Python para realizar as conversões, percebeu-se portanto que o SWIG é apenas um intermediário entre um programa C e a API da linguagem suportada.

Capítulo 6

Testes com o Cython usando funções matemáticas

Os testes realizados se baseiam em funções matemáticas simples conforme código exemplo descrito em [Behnel et al., 2017b].

O teste consiste em executar a função *integrate_f()* em loop 10.000.000 vezes com a mesma instância e assim obter um tempo médio de execução.

Foram feitas duas análises. Na primeira foi feita a medição do tempo total de execução do loop usando a função *time* do Python. Na segunda foi utilizada a ferramenta de perfil determinístico *cProfile* de [Python Software Foundation, 2017e] para verificar o tempo de execução por chamada de função.

O *cProfile* apenas realiza medição dentro do código Python, pois ele precisa adicionar seu código ao código analisado para poder fazer a medição de tempo por chamada de função. Código compilado pelo Cython não pode ser alterado e portanto o *cProfile* não pode medir chamadas internas de função nele.

Para a realização dos testes foi utilizado notebook com a seguinte configuração: Processador Intel Core2 Duo T6600, com dois núcleos de 2.20 GHz, 4 GB de memória RAM, sistema operacional Linux 64 bits, gcc versão 5.4.0

Os tempos de execução mostrados nos perfis sofre um overhead do *cProfile*, por esse motivo os tempos totais de execução nas duas análises divergem.

6.1 Código Python puro

Sem usar o cython ou outra forma de otimização. Os valores deste teste servirão de referência para os próximos testes.

Tempo total de execução: 11,45 segundos

Conclusão do teste: O tempo total de execução do módulo *integral.py* no cProfile é 16,882 segundos, tomando apenas soma dos tempos das funções *integrate_f()* e *f()*. Isso corresponde a 74,15% do tempo total de execução do teste.

Tabela 6.1: Análise de execução de código Python ao executar funções simples

Ordered by: internal time

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
10000000	12.700	0.000	19.061	0.000	integral.py:6(integrate_f)
30000000	4.182	0.000	4.182	0.000	integral.py:3(f)
1	3.535	3.535	22.766	22.766	teste.py:11(teste)
10000001	2.349	0.000	2.349	0.000	range
1	0.000	0.000	22.766	22.766	<string>:1(<module>)
1	0.000	0.000	0.000	0.000	method 'disable' of '_lsprof.Profiler' objects

50000004 function calls in 22.766 seconds

6.2 Código compilado pelo cython sem alteração

Foi utilizado mesmo código do teste anterior mas agora o módulo foi antes compilado usando o Cython.

Tempo total de execução: 5,95 segundos

Tabela 6.2: Análise de execução de código compilado pelo Cython sem alterações

Ordered by: internal time

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
10000000	5.140	0.000	5.140	0.000	{integral.integrate_f}
1	2.182	2.182	7.487	7.487	teste.py:11(teste)
1	0.165	0.165	0.165	0.165	{range}
1	0.000	0.000	7.487	7.487	<string>:1(<module>)
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

10000004 function calls in 7.487 seconds

Conclusão do teste: O tempo de execução do código compilado pelo Cython, mesmo sem modificações, caiu 48,03%. Neste teste as funções *integrate_f()* e *f()* já não estão mais na espaço de trabalho do Python, por isso o cProfile não foi além da chamada da função *integrate_f()*, pois a partir dali já é código C compilado.

6.3 Código compilado pelo Cython com tipagem das variáveis

Primeira otimização. Foram tipados os parâmetros de entrada das funções *f()* e *integrate_f()* para representar tipos de dados da linguagem C. Foram tipadas também váriaveis auxiliares usadas na função *integrate_f()*. O retorno das funções continuam sendo objetos do Python.

Tempo total de execução: 3,07 segundos

Tabela 6.3: Análise de execução de código com tipagem de variáveis compilado pelo Cython

Ordered by: internal time

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
10000000	2.814	0.000	2.814	0.000	{integral.integrate_f}
1	2.186	2.186	5.161	5.161	teste.py:11(teste)
1	0.161	0.161	0.161	0.161	{range}
1	0.000	0.000	5.161	5.161	<string>:1(<module>)
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

10000004 function calls in 5.161 seconds

Código C gerado para a função f() sem tipagem das variáveis no teste anterior

```
1 static PyObject *__pyx_pf_8integral_f(CYTHON_UNUSED PyObject *__pyx_self,
    ↪ PyObject *__pyx_v_x) {
2   __pyx_t_1 = PyNumber_Power(__pyx_v_x, __pyx_int_2, Py_None);
3   if (unlikely(!__pyx_t_1)) __PYX_ERR(0, 5, __pyx_L1_error)
4   __pyx_t_2 = PyNumber_Subtract(__pyx_t_1, __pyx_v_x);
5   if (unlikely(!__pyx_t_2)) __PYX_ERR(0, 5, __pyx_L1_error)
```

Código C gerado para a função f() com tipagem das variáveis neste teste

```
1 static PyObject *__pyx_pf_8integral_f(CYTHON_UNUSED PyObject *__pyx_self,
    ↪ double __pyx_v_x) {
2   __pyx_t_1 = PyFloat_FromDouble((pow(__pyx_v_x, 2.0) - __pyx_v_x));
3   if (unlikely(!__pyx_t_1)) __PYX_ERR(0, 5, __pyx_L1_error)
```

Conclusão do teste: Com a tipagem das variáveis o tempo reduziu mais 45,25%, em relação ao teste anterior. Pode-se ver pelos códigos C gerados que, quando as variáveis envolvidas em uma operação estão tipadas com tipos estáticos, o Cython deixa de usar funções do Python para usar funções do próprio C e somente no final ele converte para o tipo dinâmico para servir de retorno à função.

6.4 Código compilado pelo Cython com tipagem das variáveis e funções

Segunda otimização. Além das otimizações realizadas no teste anterior, agora o retorno da função *f()* é do tipo double de C e a função *f()* não está mais acessível a partir do código Python porque foi usado **cdef**. Isso não afeta o funcionamento porque ela é usada apenas pela função *integrate_f()* que consegue acessá-la pelo código C.

Tempo total de execução: 1,92 segundo

Tabela 6.4: Análise de execução de código com tipagem de variáveis e funções compilado pelo Cython

Ordered by: internal time

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	2.132	2.132	3.621	3.621	teste.py:11(teste)
10000000	1.321	0.000	1.321	0.000	{integral.integrate_f}
1	0.167	0.167	0.167	0.167	{range}
1	0.000	0.000	3.621	3.621	<string>:1(<module>)
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

10000004 function calls in 3.621 seconds

Código C da função f() com escopo restringido ao C

```
1  static double __pyx_f_8integral_f(double __pyx_v_x) {
2    double __pyx_r;
3    __Pyx_RefNannyDeclarations
4    __Pyx_RefNannySetupContext("f", 0);
5    __pyx_r = (pow(__pyx_v_x, 2.0) - __pyx_v_x);
6    goto __pyx_L0;
7    __pyx_L0:;
8    __Pyx_RefNannyFinishContext();
9    return __pyx_r;
10 }
```

Conclusão do teste: A função *f()*, com exceção de algumas funções de controle do Cython, se tornou uma função bem simples. O tempo de execução caiu novamente, 53,05% em relação ao teste anterior.

6.5 Conclusão

O aumento de desempenho foi acima do prometido em dois casos mas no último teste não foi possível observar o aumento de desempenho citado. Em [Behnel et al., 2017b] não é citado o modo em que os testes foram realizados. No entanto o aumento de 12,77 vezes na velocidade observada do programa e com poucas alterações em seu código ainda é algo surpreendente.

	Tempo da integrate_f()	Ganho obtido	Prometido pelo Cython
Python puro	16.882 s	—	—
Somente compilado	5.140 s	3.28	1.35
Variáveis tipadas em C	2.814 s	6	4
Váriáveis e funções tipadas em C	1.321 s	12.77	150

Capítulo 7

Testes com o Cython com a função *intersect*

Os testes consistem em executar a função *intersect* em loop 1.000.000 vezes com os mesmos parâmetros, para dessa maneira obter-se o tempo médio de execução do algoritmo.

O código da função *intersect* utiliza-se da NetworkX [NetworkX, 2017], que é uma biblioteca da linguagem Python para geração e manipulação de grafos. O alvo destes testes será apenas a otimização do código da função *intersect*.

Foi criado o arquivo **teste.py**, cujo conteúdo é o mesmo para todos os testes, nele está a função de teste. Primeiramente é executada a função *teste* e medido o tempo de execução com a função *time* do Python, com isso tem-se o tempo total real de execução. Após isso é usado o módulo *cProfile* do Python para a análise dinâmica da execução do teste por chamada de função.

O resultado da execução do *cProfile* é um arquivo binário chamado **Profile.prof** contendo as estatísticas. Este arquivo é usado pelo módulo *pstats* do Python para escrever uma versão texto dessas estatísticas.

Para a realização dos testes foi utilizado notebook com a seguinte configuração: Processador Intel Core2 Duo T6600, com dois núcleos de 2.20 GHz, 4 GB de memória RAM, sistema operacional Linux 64 bits, gcc versão 5.4.0

7.1 Sequência dos testes e resultados

7.1.1 Código Python puro

Sem uso do cython ou outra forma de otimização.

Tempo de execução: 1,652 segundo

Conclusão do teste: Com esse teste, tem-se o tempo normal de execução da função *intersect* sem otimização. É possível também ter noção da distribuição de tempo entre as funções. Descontando o tempo da função *teste*, a função *intersect* gastou 49,94% do tempo enquanto o método *has_edge* da classe *Graph* do NetworkX ficou com 35,81% do tempo. Essa distribuição de tempo afetará os resultados de alguns dos próximos testes, pois apenas a função *intersect* é alvo das otimizações.

Tabela 7.1: Análise de execução da função intersect

Ordered by: internal time

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1000000	5.112	0.000	8.757	0.000	intersect.py:4(intersect)
3000000	3.645	0.000	3.645	0.000	graph.py:980(has_edge)
1	1.404	1.404	10.187	10.187	teste.py:12(teste)
1	0.025	0.025	0.025	0.025	{range}
1	0.000	0.000	10.187	10.187	<string>:1(<module>)
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

4000004 function calls in 10.187 seconds

7.1.2 Código compilado pelo Cython sem modificações

Neste teste o código Python, sem nenhuma modificação, foi compilado pelo Cython.
Tempo de execução: 1,461 segundo

Tabela 7.2: Análise de execução da função intersect sem alterações compilada pelo Cython

Ordered by: internal time

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1000000	5.009	0.000	8.657	0.000	{intersect.intersect}
3000000	3.648	0.000	3.648	0.000	graph.py:980(has_edge)
1	1.330	1.330	10.014	10.014	teste.py:12(teste)
1	0.027	0.027	0.027	0.027	{range}
1	0.000	0.000	10.014	10.014	<string>:1(<module>)
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

4000004 function calls in 10.014 seconds

Análise sobre o código gerado

O código C gerado pelo Cython inclui o arquivo *header* do próprio Python (*Python.h*), os tipos de dados e funções usados no código gerado são do próprio Python, o que revela que após ser compilado ele se torna parte do Python como uma biblioteca, sem mediações.

A primeira parte do código revela muitas diretivas de compilação que adaptam o código para a versão do Python disponível, o sistema operacional, o compilador e se será compilado em C ou C++. A segunda parte é a versão em C do código Python compilado. A terceira e última parte contém funções auxiliares.

O Cython adiciona comentários com trechos do código Python original para indicar qual parte do código gerado corresponde ao código original.

Conclusão do teste: Houve uma redução de 11,56% no tempo de execução mesmo sem realizar modificações no código.

7.1.3 Código compilado pelo Cython com otimizações

Nos testes a seguir, o código Python sofreu modificações para que o Cython pudesse identificar pontos de otimização, onde pode ser substituídos os tipos de dados do Python por tipos estáticos do C e assim otimizar o código final. As modificações foram feitas e analisadas separadamente, foi vista de que forma o desempenho foi afetado, quais modificações o Cython efetuou baseado nas alterações feitas no código Python e se elas foram eficazes ou não.

O primeiro parâmetro da função é uma instância da classe *Graph* da biblioteca *NetworkX* que não possui nenhuma conversão para um tipo em C ou outra estrutura mais simples e portanto não integrará os testes.

Conversões automáticas

Neste teste, foram tipados o terceiro parâmetro de entrada da função *intersect* **v** e a variável interna **u**, ambas são do tipo inteiro, esse tipo possue conversão automática pelo Cython.

```
1 def intersect(G, K, int v)
2    cdef int u
3    return [u for u in K if G.has_edge(v, u)]
```

O código gerado apresentou as seguintes mudanças. As variáveis **u** e **v** deixaram de ser objetos Python que representavam um valor inteiro para serem do tipo estático *int* do C.

Teste anterior

```
1  PyObject *__pyx_v_v = 0;
2  PyObject *__pyx_v_u = NULL;
```

Teste atual

```
1  int __pyx_v_v;
2  int __pyx_v_u;
```

Ao analisar o código gerado observou-se que o Cython acabou não usando as conversões, ele converteu as variáveis de volta para objetos Python para usá-las no método *has_edge*.

```
1 807: __pyx_t_7 = __Pyx_PyObject_GetAttrStr(__pyx_v_G, __pyx_n_s_has_edge);
     ↪ // Obtém referência ao método has_egde do grafo G
2 809: __pyx_t_8 = __Pyx_PyInt_From_int(__pyx_v_v); // Converte o tipo está
     ↪ tico de volta a um inteiro do Python
3 831: PyTuple_SET_ITEM(__pyx_t_12, 0+__pyx_t_11, __pyx_t_8); // Coloca o
     ↪ inteiro "v", agora um objeto Python em uma tupla para passar como par
     ↪ âmetro
4 836: __pyx_t_5 = __Pyx_PyObject_Call(__pyx_t_7, __pyx_t_12, NULL); // E
     ↪ chama o método has_edge com o tipo inteiro Python
```

O método *Graph.has_edge()* recebe como parâmetros números inteiros do Python e não tipos estáticos do C, já que a biblioteca NetworkX não foi compilada pelo Cython, então esse método está fora do escopo do código gerado e por isso as variáveis convertidas para tipos primitivos não puderam ser usadas. Ao contrário do esperado, que era redução do tempo de execução, houve um aumento no tempo deste teste em relação ao anterior.

Tempo de execução: 1,549 segundo

Conclusão do teste: Mudar as variáveis **u** e **v** da função *intersect* do tipo inteiro do Python para o tipo estático *int* não trouxe impactos positivos. Como existe um escopo dentro do qual as variáveis convertidas **cdef** pode existir, e como as variáveis **u** e **v** seriam usadas por um método de uma classe Python que está fora desse escopo, por esse motivo o Cython converteu as variáveis novamente, o que causou queda no desempenho.

Tabela 7.3: Análise de execução da função intersect com tipagem de variáveis compilada pelo Cython

Ordered by: internal time

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1000000	5.052	0.000	8.697	0.000	{intersect.intersect}
3000000	3.645	0.000	3.645	0.000	graph.py:980(has_edge)
1	1.315	1.315	10.038	10.038	teste.py:12(teste)
1	0.026	0.026	0.026	0.026	{range}
1	0.000	0.000	10.038	10.038	<string>:1(<module>)
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

4000004 function calls in 10.038 seconds

Conversões não automáticas

Nos testes a seguir foi feita a tipagem estática do segundo parâmetro da função *intersect*. O intuito é converter um tipo *list* do Python para um vetor de inteiros (*int* *).

Definição do parâmetro como *int* * diretamente

Tipou-se o segundo parâmetro da função *intersect* como um ponteiro de inteiros para verificar se o Cython realizaria a conversão ou não.

```
def intersect(G, int *K, int sizeK, v):
```

Foi inserido mais um parâmetro para indicar o número de elementos da lista K.
No entanto, o compilador retornou o seguinte erro.

```
intersect.pyx:5 Cannot convert Python object argument to type 'int *'
```

Conclusão do teste: Observou-se que não há conversão automática de um tipo *list* do Python para o tipo *int* * em C.

Uso de Python Arrays com memory views

O Python possui um módulo interno que suporta arrays uni-dimensionais de tipos simples como caracteres, inteiros e números de ponto flutuante. É possível acessar o *array* em C a partir de uma *list* Python com o Cython.

Memory view é uma forma de acesso segura os dados internos de um objeto Python. [Behnel et al., 2017c]

```
1 from cpython cimport array
2 import array
3
4 def intersect(G, K, v):
5       cdef array.array a = array.array('i', K)
6       cdef int[:] u = a
```

Tempo de execução: 10,124 segundos

Conclusão do teste: Foi possível acessar a *list* K mas o overhead gerado era muito grande. Não só o tempo aumentou como também o número de linhas do código C gerado neste teste. Não houve vantagem em desempenho nessa abordagem.

Tabela 7.4: Análise de execução da função intersect compilada pelo Cython com uso de memory views

Ordered by: internal time

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1000000	13.786	0.000	17.875	0.000	{intersect.intersect}
3000000	4.089	0.000	4.089	0.000	graph.py:980(has_edge)
1	1.602	1.602	19.504	19.504	teste.py:12(teste)
1	0.027	0.027	0.027	0.027	{range}
1	0.000	0.000	19.504	19.504	<string>:1(<module>)
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

4000004 function calls in 19.504 seconds

Python Arrays com acesso direto ao ponteiro C

Para evitar-se o overhead do teste anterior, neste teste acessou-se diretamente os dados do objeto *array* do Python. Esse acesso não é seguro por não possuir checagem de tipo ou limite e se for usado para escrita pode gerar erros.

Tempo de execução: 1,964 segundo

Tabela 7.5: Análise de execução da função intersect compilada pelo Cython com acesso direto aos dados em C

Ordered by: internal time

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1000000	5.593	0.000	9.249	0.000	{intersect.intersect}
3000000	3.656	0.000	3.656	0.000	graph.py:980(has_edge)
1	1.312	1.312	10.586	10.586	teste.py:12(teste)
1	0.026	0.026	0.026	0.026	{range}
1	0.000	0.000	10.586	10.586	<string>:1(<module>)
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

4000004 function calls in 10.586 seconds

Conclusão do teste: Apesar de obter-se um desempenho melhor que o teste anterior com Python Array e memory views, o tempo de execução ficou ainda ligeramente maior que o do teste com Cython sem otimizações. Mesmo após retirar as memory views, ainda houve algum overhead causado pelo o uso do *Python array*.

7.2 Conclusão

O melhor desempenho do Cython com o código da função *intersect* foi uma redução de 11,56% no tempo de execução total no teste em que o código, sem alterações, foi compilado.

É possível conseguir mais otimizações com o Cython através do uso de tipagem estática de variáveis, porém no caso da função *intersect* não foi possível porque:

1. Algumas variáveis em que mudou-se o tipo de objeto python para um tipo estático da linguagem C seriam usadas na chamada de um método externo ao código compilado com o Cython e que recebia como parâmetros objetos Python e por isso necessitariam serem convertidas de volta;

2. Não havia conversão automática de tipo de dado e a conversão manual demostrou um desempenho pior;

3. Não há conversão disponível para certos tipos de dados, como o caso do parâmetro **G** que é instância da classe *Graph* do networkx.

Tabela 7.6: Comparativo de tempos dos testes com o Cython

	Tempo total de execução da *intersect()*	Ganho ou perda de desempenho
Código Python puro	1,652	—
Código compilado sem modificações	1,461 s	- 11,56%
Conversões automáticas	1,549 s	- 6,23%
Conversões não automáticas com Python array e memoryview	10,124 s	+ 512,83%
Conversões não automáticas com Python array com acesso ao ponteiro C	1,964 s	+ 18,88 %

Capítulo 8

Teste do Ctypes com funções matemáticas

Usou-se a função de cálculo de integral descrita na documentação do Cython disponível em [Behnel et al., 2017b]. Foram feitas duas implementações, uma em linguagem C para teste com o módulo ctypes e outra em Python para efeito de comparação.

O teste consiste em executar a função *integrate_f()* em loop 10.000.000 vezes com a mesma instância e assim obter um tempo médio de execução.

Foram feitas duas análises. Na primeira foi feita a medição do tempo total de execução do loop usando a função *time* do Python. Na segunda foi utilizada a ferramenta de perfil determinístico *cProfile* de [Python Software Foundation, 2017e] para verificar o tempo de execução por chamada de função.

Para a realização dos testes foi utilizado notebook com a seguinte configuração: Processador Intel Core2 Duo T6600, com dois núcleos de 2.20 GHz, 4 GB de memória RAM, sistema operacional Linux 64 bits, gcc versão 5.4.0

8.1 Resultado dos testes

8.1.1 Python puro

Tempo de execução: 16,91 segundos

Tabela 8.1: Análise de execução de funções no Python

Ordered by: internal time

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
10000000	64.197	0.000	111.226	0.000	teste_python_puro.py:7(integrate_f)
30000000	34.827	0.000	34.827	0.000	teste_python_puro.py:4(f)
1	13.492	13.492	125.032	125.032	teste_python_puro.py:15(teste)
10000001	12.516	0.000	12.516	0.000	{range}
1	0.000	0.000	125.032	125.032	<string>:1(<module>)
1	0.000	0.000	0.000	0.000	{method 'disable'of '_lsprof.Profiler'objects}

50000004 function calls in 125.032 seconds

8.1.2 Biblioteca em C importada com ctypes

Tempo de execução: 8,34 segundos

Tabela 8.2: Análise de execução de funções escritas em C acessadas pelo módulo ctypes

Ordered by: internal time

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	8.029	8.029	8.344	8.344	teste_ctypes.py:10(teste)
1	0.315	0.315	0.315	0.315	{range}
1	0.000	0.000	8.344	8.344	<string>:1(<module>)
1	0.000	0.000	0.000	0.000	{method 'disable'of '_lsprof.Profiler'objects}

4 function calls in 8.34 seconds

8.2 Análise dos testes

Tabela 8.3: Comparação entre os tempos do ctypes com a versão Python puro.

	Tempo de execução	Redução no tempo
Python puro	16,912 s	—
ctypes	8,337 s	50,70%

O uso do módulo ctypes para acessar as funções *integrate_f* e f escritas em C trouxe uma redução de mais de metade do tempo de execução.

8.3 Conclusão

O módulo ctypes trabalha como uma interface entre o código Python e o código C compilado, de uma maneira semelhante ao SWIG em sua forma de trabalhar, com a diferença que esse é uma ferramenta nativa do Python e faz apenas a interface entre Python e C.

Contudo o ctypes possui uma vantagem sobre o SWIG, ele pode utilizar bibliotecas dinâmicas já prontas, inclusive as bibliotecas padrão do Linux ou Windows. No SWIG é necessário compilar funções wrappers junto com a biblioteca para poder utilizá-la.

O ganho de desempenho do módulo ctypes é representativo porém o esforço em escrever códigos Python para C pode representar um desafio, diferente do Cython que consegue converter código Python em código C, desde que não haja funcionalidades não suportadas.

O Cython também permite acessar bibliotecas escritas originalmente em C a partir de código compilado por ele.

Capítulo 9

Conclusão

Das três ferramentas apresentadas, a que se mostrou de mais fácil uso e que conseguiu reescrever sozinha o código Python da função *intersect* foi o Cython.

No caso da solução desenvolvida por [Züge, 2012], esta faz uso da biblioteca [NetworkX, 2017]. Como observou-se no teste no capítulo 7 na seção 7.1.3, a função *intersect*, compilada pelo Cython, chama o método *has_edge* da classe *Graph* da NetworkX que não foi compilada pelo Cython, com isso saiu-se de um ambiente em que o código era executado diretamente pelo sistema operacional para o ambiente intermediado pelo interpretador Python, fato que afetou o desempenho do código compilado pelo Cython.

Para melhorar o desempenho do Cython com a função *intersect* uma possível solução seria compilar com o Cython todos os arquivos da biblioteca NetworkX.

O módulo ctypes do Python, apesar de seu bom desempenho, possui a limitação de não permitir conversões de tipo de dados antes das chamadas de funções, o que limita o seu uso a apenas tipos de dados simples semelhantes nas duas linguagens nos quais há conversão automática.

O SWIG permite a conversão não automática de tipos de dados utilizando os *typemaps*, onde insere-se o código de conversão de tipo definido pelo desenvolvedor.

O SWIG e o ctypes não realizam conversão de código Python para C como o Cython.

9.1 Trabalhos futuros

Realizar a compilação dos códigos da biblioteca NetworkX usando o Cython.

Implementar as rotinas críticas da solução de [Züge, 2012] usando a API da linguagem Python.

Referências Bibliográficas

[Behnel et al., 2017a] Behnel, S., Bradshaw, R., Seljebotn, D. S., Ewing, G., Stein, W. e Gellner, G. (2017a). Cython - using c libraries. http://docs.cython.org/en/latest/src/tutorial/clibraries.html. Acessado em 04/07/2017.

[Behnel et al., 2017b] Behnel, S., Bradshaw, R., Seljebotn, D. S., Ewing, G., Stein, W. e Gellner, G. (2017b). Faster code via static typing. http://docs.cython.org/en/latest/src/quickstart/cythonize.html. Acessado em 19/06/2017.

[Behnel et al., 2017c] Behnel, S., Bradshaw, R., Seljebotn, D. S., Ewing, G., Stein, W. e Gellner, G. (2017c). Working with python arrays - safe usage with memory views. http://docs.cython.org/en/latest/src/tutorial/array.html#safe-usage-with-memory-views. Acessado em 19/06/2017.

[Ewing, 2017] Ewing, G. (2017). Pyrex - a language for writing python extension modules. http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/. Acessado em 19/06/2017.

[NetworkX, 2017] NetworkX (2017). Networkx. https://networkx.github.io/. Acessado em 19/06/2017.

[(PGI/JCNS-TA), 2017] (PGI/JCNS-TA), D. (2017). Using c from python: How to create a ctypes wrapper. https://pgi-jcns.fz-juelich.de/portal/pages/using-c-from-python.html. Acessado em 19/06/2017.

[Project Jupyter, 2017] Project Jupyter (2017). Project jupyter support interactive data science and scientific computing across all programming languages. http://jupyter.org/. Acessado em 19/06/2017.

[Python Software Foundation, 2017a] Python Software Foundation (2017a). Complex number objects. https://docs.python.org/3/c-api/complex.html. Acessado em 19/06/2017.

[Python Software Foundation, 2017b] Python Software Foundation (2017b). Ctypes — a foreign function library for python. https://docs.python.org/3/library/ctypes.html. Acessado em 19/06/2017.

[Python Software Foundation, 2017c] Python Software Foundation (2017c). distutils — building and installing python modules. https://docs.python.org/3.6/library/distutils.html. Acessado em 19/06/2017.

[Python Software Foundation, 2017d] Python Software Foundation (2017d). Implementação principal da linguagem de programação python, escrita em linguagem c. https://github.com/python/cpython/. Acessado em 19/06/2017.

[Python Software Foundation, 2017e] Python Software Foundation (2017e). The python profilers. https://docs.python.org/2/library/profile.html. Acessado em 19/06/2017.

[SageMath, 2017] SageMath (2017). Sagemath is a free open-source mathematics software system. http://www.sagemath.org/. Acessado em 19/06/2017.

[SWIG, 2017] SWIG (2017). Typemaps multi argument typemaps. http://www.swig.org/Doc2.0/SWIGDocumentation.html#Typemaps_multi_argument_typemaps. Acessado em 19/06/2017.

[Züge, 2012] Züge, A. P. (2012). Solução exata do problema da clique máxima. Dissertação de Mestrado, UFPR.