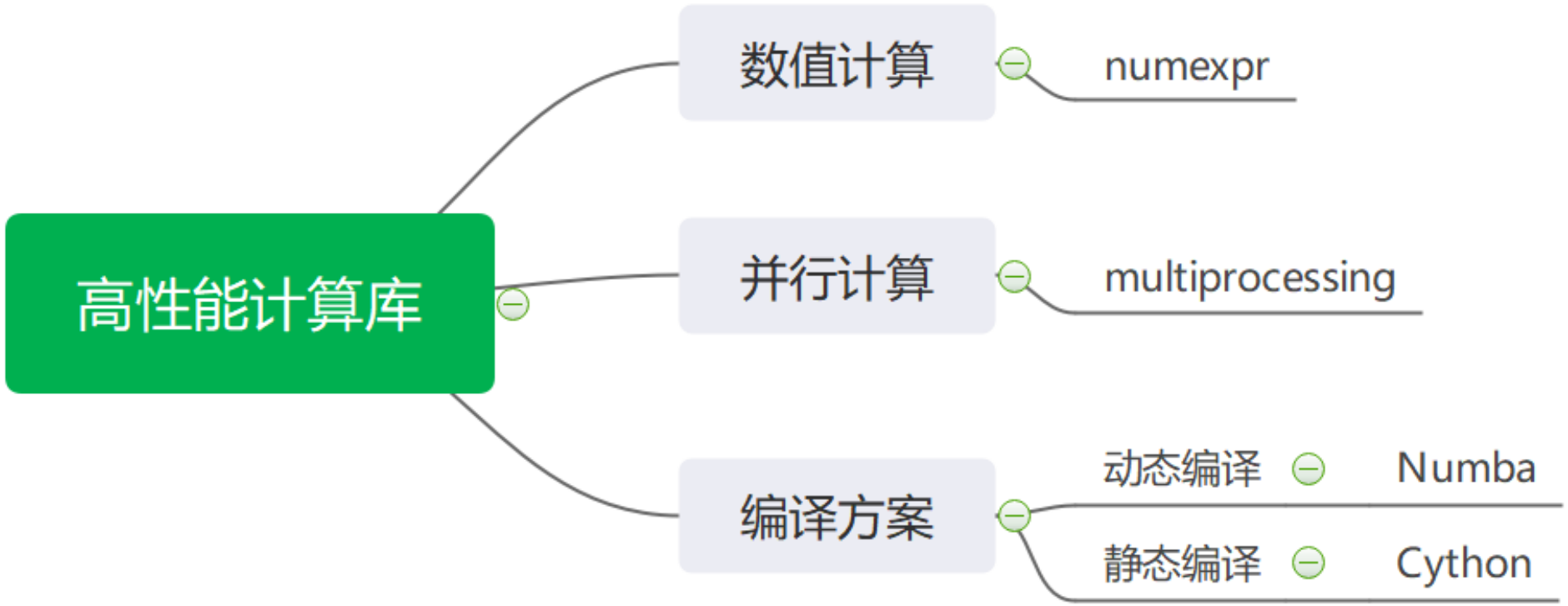
Python

高性能计算库

智能系统实验室

清华大学iCenter

导学


高性能计算库
数值计算
numexpr
并行计算
multiprocessing
编译方案
动态编译
Numba
静态编译
Cython

常用高性能库

- numexpr，用于快速数值运算。
- multiprocessing，python内建的并行处理模块。
- Numba，用于为CPU动态编译Python代码。
- Cython，用于合并Python和C语言静态编译泛型。

数值计算加速

numxepr

Python 数值计算

例如我们想要在包含 50 万个数值的数组上求取某个复杂数学表达式的值：

$$y = \sqrt{|\cos(x)|} + \sin(2 + 3x)$$

In [3]:
```
from math import *
def f(x):
    return abs(cos(x)) ** 0.5 + sin(2 + 3 * x)
```

In [4]:
```
I = 500000
a_py = range(I)
```

1. 包含显式循环的标准Python函数

In [5]:
```
def f1(a):
    res = []
    for x in a:
        res.append(f(x))
    return res
```

2. 包含隐含循环的送代子方法（list推导式）

In [6]:
```
def f2(a):
    return [f(x) for x in a]
```

3. 包含隐含循环、使用eval的迭代子方法

In [7]:
```
def f3(a):
    ex = 'abs(cos(x)) ** 0.5 + sin(2 + 3 * x)'
    return [eval(ex) for x in a]
```

eval() 函数用来执行一个字符串表达式，并返回表达式的值。

4. NumPy向量化实现

In [8]:
```
import numpy as np
```

In [9]:
```
a_np = np.arange(I)
```

In [10]:
```
def f4(a):
    return (np.abs(np.cos(a)) ** 0.5 +
            np.sin(2 + 3 * a))
```

5. numexpr单线程实现

In [11]:
```
import numexpr as ne
```

In [12]:
```
def f5(a):
    ex = 'abs(cos(a)) ** 0.5 + sin(2 + 3 * a)'
    ne.set_num_threads(1)
    return ne.evaluate(ex)
```

6. numexpr多线程实现

In [13]:
```
def f6(a):
    ex = 'abs(cos(a)) ** 0.5 + sin(2 + 3 * a)'
    ne.set_num_threads(16)
    return ne.evaluate(ex)
```

In [21]:
```
perf_comp_data(func_list, data_list)
```

```
function: f6, av. time sec:   0.00561, relative:    1.0
function: f5, av. time sec:   0.01448, relative:    2.6
function: f4, av. time sec:   0.01851, relative:    3.3
function: f2, av. time sec:   0.35237, relative:   62.8
function: f1, av. time sec:   0.38011, relative:   67.7
function: f3, av. time sec:   6.78153, relative: 1208.7
```

最快的是f6和f5，f6的速度优势取决于可用核心数量，然后是向量化Numpy版本f4，之后是纯python实现的版本，最慢的是f3，因为对这样大量的求值运算使用eval会造成巨大的负担。
在numexpr的例子中，基于字符串的表达式计算一次之后被编译供以后使用；而使用Python eval函数，这一操作要进行50万次。

注意观察运行时间 [4]: 数字变为星号代表正在运行

想一想，练一练

需要安装库pip install numpexpr

- 在jupyter notebook中运行以下代码

```
import numpy as np
import numexpr as ne
nx, ny = 1200, 1500
a = np.linspace(0.,3.1416,nx*ny).reshape(nx,ny)
for i in range(100):
      b = np.sin(a+i)**2 + np.cos(a+i)**2 + a**1.5
```

把最后一行改为

```
      b = ne.evaluate("sin(a+i)**2 + cos(a+i)**2 + a**1.5")
```

再运行一次

并行计算加速

multiprocessing

并行计算

- 模拟几何布朗运动
- 使用multiprocessing库

In [36]:
```
import multiprocessing as mp
```

In [37]:
```
import math
def simulate_geometric_brownian_motion(p):
    M, I = p
      # time steps, paths
    S0 = 100; r = 0.05; sigma = 0.2; T = 1.0
      # model parameters
    dt = T / M
    paths = np.zeros((M + 1, I))
    paths[0] = S0
    for t in range(1, M + 1):
        paths[t] = paths[t - 1] * np.exp((r - 0.5 * sigma ** 2) * dt +
                    sigma * math.sqrt(dt) * np.random.standard_normal(I))
    return paths
```

In [38]:
```
paths = simulate_geometric_brownian_motion((5, 2))
paths
```

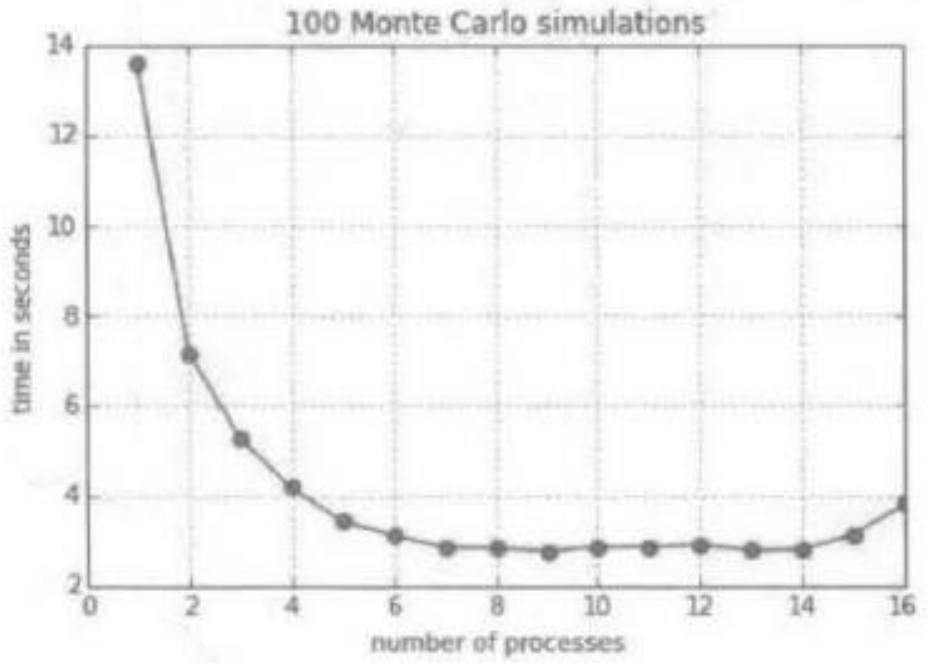
Out[38]:
```
array([[ 100.        ,  100.        ],
       [ 107.75994389,   96.38974629],
       [ 111.36869054,  104.04967034],
       [  97.36585627,   99.81282855],
       [ 100.33598469,   85.5209064 ],
       [  88.58477948,   76.08336793]])
```

- 在一个具有16个核心的服务器上进行100次模拟。

[39]:
```
I = 10000  # number of paths
M = 100  # number of time steps
t = 100  # number of tasks/simulations
```

[40]:
```
# running on server with 16 cores
from time import time
times = []
for w in range(1, 17):
    t0 = time()
    pool = mp.Pool(processes=w)
      # the pool of workers
    result = pool.map(simulate_geometric_brownian_motion, t * [(M, I), ])
      # the mapping of the function to the list of parameter tuples
    times.append(time() - t0)
```

[42]:
```
plt.plot(range(1, 17), times)
plt.plot(range(1, 17), times, 'ro')
plt.grid(True)
plt.xlabel('number of processes')
plt.ylabel('time in seconds')
plt.title('%d Monte Carlo simulations' % t)
```



可以看出性能和可用核心数量成正比。

编译加速方案

Numba Cython

动态编译

- python中通常会影响性能的问题：包含嵌套循环的算法

In [43]:
```
from math import cos, log
def f_py(I, J):
    res = 0
    for i in range(I):
        for j in range (J):
            res += int(cos(log(1)))
    return res
```

In [44]:
```
I, J = 2500, 2500
%time f_py(I, J)
```

```
CPU times: user 2.39 s, sys: 17.2 ms, total: 2.41 s
Wall time: 2.41 s
```

Out[44]:
```
6250000
```

使用NumPy实现向量化

```
In [45]: def f_np(I, J):
             a = np.ones((I, J), dtype=np.float64)
             return int(np.sum(np.cos(np.log(a)))), a

In [46]: %time res, a = f_np(I, J)
         CPU times: user 117 ms, sys: 67.3 ms, total: 185 ms
         Wall time: 183 ms

In [47]: a.nbytes
Out[47]: 50000000
```

虽然速度变快，但并不能高效利用内存，占用内存达到50MB，当i和j足够大，Numpy方法就会不可行。

调用Numba中jit函数

```
In [48]: import numba as nb

In [49]: f_nb = nb.jit(f_py)

In [50]: %time f_nb(I, J)
CPU times: user 113 ms, sys: 8.23 ms, total: 121 ms
Wall time: 120 ms

Out[50]: 6250000
```

jit可以编译部分代码，对于它能够编译的代码，将它们转换为函数，并编译成机器码。然后将其余部分的代码提供给 python 解释器。

```
In [51]: func_list = ['f_py', 'f_np', 'f_nb']
         data_list = 3 * ['I, J']
```

```
In [52]: perf_comp_data(func_list, data_list)
```

```
function: f_nb, av. time sec:   0.00000, relative:    1.0
function: f_np, av. time sec:   0.13171, relative: 82319.1
function: f_py, av. time sec:   2.20040, relative: 1375253.4
```

嵌套循环实现的Numba版本是目前最快的方案，甚至远快于NumPy向量化版本。Python版本比其他两种版本慢得多。

Numba的优势

- 效率：使用Numba只需要花费很少的额外精力。原始函数往往完全不需要修改；需要做的就是调用jit函数。
- 加速：Numba的执行速度显著提高，不仅和纯Python相比是如此，即使对向量化的NumPy实现也有明显优势。
- 内存：使用Numba不需要初始化大型数组对象；编译器专门为手上的问题生成机器代码（和NumPy的"通用"函数相比）并维持和纯Python相同的内存效率。

用Cython进行静态编译

- 嵌套循环算法

和上一个嵌套循环示例相比，这次内循环次数由外循环次数放大。

In [68]:
```
def f_py(I, J):
    res = 0.  # we work on a float object
    for i in range(I):
        for j in range (J * I):
            res += 1
    return res
```

In [69]:
```
I, J = 500, 500
%time f_py(I, J)
```

```
CPU times: user 7.73 s, sys: 66.2 ms, total: 7.8 s
Wall time: 7.84 s
```

Out[69]:
```
125000000.0
```

使用 Cython 静态类型声明的嵌套循环

In [76]:
```
%%cython
#
# Nested loop example with Cython
#
def f_cy(int I, int J):
    cdef double res = 0
    # double float much slower than int or long
    for i in range(I):
        for j in range (J * I):
            res += 1
    return res
```

通过pyximport模块从Cython导入

In [70]:
```
import pyximport
pyximport.install()
```

Out[70]:
```
(None, <pyximport.pyximport.PyxImporter at 0x1170e9470>)
```

In [71]:
```
import sys
sys.path.append('data/')
  # path to the Cython script
  # not needed if in same directory
```

In [72]:
```
from nested_loop import f_cy
```

In [73]:
```
%time res = f_cy(I, J)
```

```
CPU times: user 203 ms, sys: 2.87 ms, total: 206 ms
Wall time: 205 ms
```

In [74]:
```
res
```

Out[74]:
```
125000000.0
```

小结

- 计算库

  针对不同问题有相应的计算加速方案，例如numexpr。

- 并行化

  有些 Python 并行化功能，可以加速程序的计算时间，例如multiprocessing。

- 编译方案

  有一些强大的动态或静态编译解决方案，如Cython和Numba。

参考

- [德] 伊夫·希尔皮斯科（Yves Hilpisch） 著，姚军 译，Python金融大数据分析，人民邮电出版社，2015.
- Numexpr文档：http://github.com/pydata/numexpr。
- Numba文档：http://github.com/numba/numba。
- Cython首页：http://cython.org。
- Numbapro文档：http://docs.continuum.io/numbapro。

谢谢指正！