Optimisation CPU avec Python

Xavier Juvigny

ONERA/HPC

Décembre 2013

Principe d’optimisation CPU

- Il arrive souvent que dans un code, une petite partie du code prends 90% du temps CPU
- Il est inutile de tenter d’optimiser tout le code !
- Cette partie de code rallientissant l’application n’est pas toujours celle qu’on croit !
- Il faut utiliser des outils spécialisés pour détecter sans *a priori* les points rallentissant l’application
- Il suffit alors de chercher à optimiser ces parties de code qu’on aura détecté à l’aide d’outils.

Outils de profiling CPU

Python est livré par défaut avec trois modules permettant le profiling d'applications Python :

1. `cProfile`, écrit en C, qui permet soit d'afficher des statistiques basiques de profiling, soit de générer un fichier binaire contenant les statistiques issues d'un appel à une fonction, illisible par un humain !
2. `profile` écrit un pure python ayant les mêmes fonctionnalités que le précédent, mais avec un coût plus grand que le précédent, mais permettant d'écrire une extension au profiler.
3. `pstats` permet de lire le fichier généré et de sortir des statistiques un peu plus fines sur le temps pris par les fonctions

```
import cProfile
import StaticHeatEquation_naive as StaticHeatEquation

cProfile.run('StaticHeatEquation.solve(300)')
```

Outils de profiling CPU...

```
import cProfile
import StaticHeatEquation_naive as StaticHeatEquation

cProfile.run('StaticHeatEquation.solve(300)')
```

```
      761286 function calls in 47.838 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000   47.838   47.838 <string>:1(<module>)
     1    0.112    0.112    0.130    0.130 StaticHeatEquation_naive.py:110(saveSol)
     1    0.002    0.002   47.837   47.837 StaticHeatEquation_naive.py:119(solve)
     1    0.026    0.026    0.032    0.032 StaticHeatEquation_naive.py:18(applyDirichlet)
     1    0.001    0.001    0.001    0.001 StaticHeatEquation_naive.py:3(buildGrid)
   316   17.324    0.055   17.480    0.055 StaticHeatEquation_naive.py:41(prodMatVect)
   631    8.687    0.014    8.966    0.014 StaticHeatEquation_naive.py:58(dot)
   945   20.748    0.022   21.174    0.022 StaticHeatEquation_naive.py:70(axpby)
     1    0.051    0.051   47.672   47.672 StaticHeatEquation_naive.py:80(GC)
  1895    0.001    0.000    0.001    0.000 {len}
   604    0.000    0.000    0.000    0.000 {math.fabs}
 91808    0.006    0.000    0.006    0.000 {method 'append' of 'list' objects}
     1    0.000    0.000    0.000    0.000 {method 'close' of 'file' objects}
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects
 90300    0.017    0.000    0.017    0.000 {method 'write' of 'file' objects}
     1    0.000    0.000    0.000    0.000 {open}
574778    0.863    0.000    0.863    0.000 {range}
```

Autres fonctionnalités de cProfile

Il est possible d'utiliser directement le profiler sans écrire un script via la ligne de commande :

`python -m cProfile StaticHeatEquation_naive.py`

On peut écrire les statistiques dans un fichier binaire :

- Via un script :

```
import cProfile
import StaticHeatEquation_naive as StaticHeatEquation

cProfile.run('StaticHeatEquation.solve(300)','prof_stats'
    )
```

- Via la ligne de commande :

  `python -m cProfile -o prof_stats StaticHeatEquation_naive.py`

Utilisation de pstats

Le module pstats propose divers outils pour exploiter les statistiques de profiling générer par le module cProfile

```
import pstats
p = pstats.Stats('prof_stats')
p.strip_dirs().sort_stats(-1).print_stats()
```

où la méthode strip_dirs enlève les chemins des modules et sort_stats trie les fonctions par ordre alphabétique.

```
p.sort_stats('time').print_stats(10)
```

va trier les fonctions par temps propre passé décroissant et afficher les dix fonctions les plus coûteuses.

```
p.print_callers(0.5, 'run')
```

va afficher la première moitié des fonctions appelant la méthode run.

Autres outils pour optimiser python

Pour expérimenter des petits bouts de code pour voir lequel est le plus efficace, Python propose le module `timeit`.

Exemple :

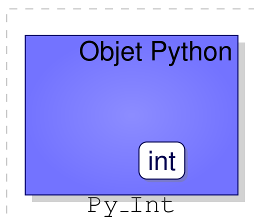
```
$ python -m timeit '"-".join(str(n) for n in range(100))'
10000 loops, best of 3: 40.3 usec per loop
$ python -m timeit '"-".join([str(n) for n in range(100)])'
10000 loops, best of 3: 33.4 usec per loop
$ python -m timeit '"-".join(map(str, range(100)))'
10000 loops, best of 3: 25.2 usec per loop
```

qu'on peut écrire sous forme de script :

```python
import timeit
timeit.timeit('"-".join(str(n) for n in range(100))', number
    =10000)
timeit.timeit('"-".join([str(n) for n in range(100)])', number
    =10000)
timeit.timeit('"-".join(map(str, range(100)))', number=10000)
```

Pourquoi Python est-il lent ?

- Python ne considère que des objets. Un simple entier est lui-même un objet !



- Python doit analyser l'objet pour vérifier son type, puis aller chercher la valeurs de l'entier à l'intérieur de l'objet, et ceci à chaque accès de cet objet !

for i in range...versus for i in xrange...

- range créée une liste d'entier qu'on parcourt ensuite pour la boucle
- xrange génère un objet qui génère des entiers successifs au fur et à mesure des besoins.
- xrange est plus rapide et plus léger en mémoire que range

```
$ python -m timeit '"-".join([str(n) for n in range(200)])'
10000 loops, best of 3: 37.8 usec per loop
$ python -m timeit '"-".join([str(n) for n in xrange(200)])'
10000 loops, best of 3: 36.3 usec per loop
```

Concaténation versus extension

- La concaténation consiste à créer une liste vide et de rajouter les éléments au fur et à mesure dans une boucle
- L’extension consiste à créer directement une liste contenant les éléments décrits par une boucle

Exemple :

```
$ python -m timeit -s "l=[]" "for i in xrange(100):" "  l.append(i)"
100000 loops, best of 3: 11.5 usec per loop
$ python -m timeit 'l = [i for i in xrange(100)]'
100000 loops, best of 3: 3.98 usec per loop
```

- Dans la concaténation, python doit reévaluer à chaque fois la liste puis accéder au dernier élément créé
- Dans l’extension, il construit un par un les éléments de la liste sans la réévaluer et sans devoir accèder au dernier élément créé.

Liste contre dictionnaire

A priori, pas prévu pour les mêmes usages, mais grâce à la versalité de Python, un dictionnaire peut être très bien remplacé par une liste !

```
$ python -m timeit 'lst_carre = [ (x,x*x) for x in xrange(1000) ]'
10000 loops, best of 3: 92.4 usec per loop
$ python -m timeit 'dic_carre = { x : x*x for x in xrange(1000) }'
10000 loops, best of 3: 94.7 usec per loop
```

Pour trouver un élément dans la liste par rapport à une clef :

```python
a = filter( lambda x : x[0] == 101, lst_carre)[0][1]
```

mais il est préférable si on doit accéder souvent à des éléments via leur clefs d'associer une fonction de hashage…