Mesurer les performances (CPU) sous Linux

Résumé :

Il existe des outils permettant de tracer les temps CPU utilisés (*profiling*).

Sur Linux, on utilise l'outil `gprof`. L'usage de cet outil impose de compiler tous les sources avec l'option "`-pg`". Le sur-coût de l'instrumentation est négligeable. La résultat du *profiling* est un fichier texte qu'il faut interpréter.

Pour simplifier l'interprétation, on propose en fin de document un outil pour tracer un graphe à partir du fichier texte produit par le calcul « profilé ».

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

Table des matières

1 Gprof....3
1.1 Construction d'une version avec gprof....3
1.2 Sur-coût de l'instrumentation....3
1.3 Exécution de l'exécutable instrumenté avec waf....3
1.4 Exécution de l'exécutable instrumenté avec astk....4
1.5 Exploitation des résultats....4
1.6 Dépouiller les résultats du profiling....4
1.7 Analyse de l'exemple....6
1.8 Génération d'un graphe avec gprof2dot....6
1.8.1 Présentation....6
1.8.2 Utilisation....7

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

1 Gprof

1.1 Construction d'une version avec gprof

Pour fonctionner, gprof a besoin que les sources aient été compilées avec l'option `-pg`.

Pour conserver la construction par défaut de Code_Aster (sans les options de gprof), on utilise un alias `waf_prof` (il suffit de créer un lien symbolique pour cela).

Ensuite, il suffit d'inclure la configuration `gprof` après la configuration de la machine pour ajouter les options nécessaires.

Sur les machines Calibre et les serveurs officiels, l'option `--use-config` est facultative en temps normal (déterminée automatiquement quand l'environnement `env_unstable.sh` est chargé). Ici, nous sommes obligés de la nommer explicitement. Remplacez donc XXX par `calibre7` ou `aster5` ou...

```
cd $HOME/dev/codeaster/src
ln -s waf_variant waf_prof
./waf_prof configure --use-config=XXX,gprof --prefix=../install/prof
./waf_prof install
```

Remarque

Le mode optimisé est a priori préférable pour mesurer les "vraies" performances du code. En revanche, le mode "`debug`" est nécessaire si l'on veut connaître les lignes les plus consommatrices (utilisez alors `./waf_prof intall_debug`).
On a malheureusement observé un problème inexplicable en mode "`debug`" : le résultat du `profiling` indiquait des liens d'appel entre routines qui n'existaient pas ! On peut toutefois espérer que cette anomalie n'invalide pas entièrement le reste de la mesure.

1.2 Sur-coût de l'instrumentation

A titre d'exemple, sur le test `ssnv506c`, on a obtenu les résultats globaux suivants :

* en mode `nodebug` sans instrumentation	138s
* en mode `nodebug` avec instrumentation	139s
* en mode `debug` sans instrumentation	218s
* en mode `debug` avec instrumentation	228s

On constate que l'instrumentation a un coût CPU négligeable.

1.3 Exécution de l'exécutable instrumenté avec waf

Pour lancer un test avec `waf`, on peut ajouter une ligne de ce type dans le fichier `.export` pour récupérer le fichier de `gprof` et le copier, par exemple, dans `/tmp` :

```
F nom /tmp/gmon.out R 0
```

Il suffit ensuite de lancer : `./waf_prof test -n ssnv506c`

Voir le paragraphe suivant pour l'utilisation de ce fichier `gmon.out`.

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

1.4 Exécution de l'exécutable instrumenté avec astk

On peut également exécuter l'étude que l'on veut "profiler" avec l'exécutable instrumenté dans astk.
Pour cela, il faut déclarer la version instrumentée dans la liste des versions utilisables localement.
Dans le fichier $HOME/.astkrc/prefs, ajouter la ligne :

```
vers : DEV_PROF:$HOME/dev/codeaster/install/prof
```

Dans astk, il suffit ensuite de sélectionner la version nommée DEV_PROF.

L'exécution de l'étude produit un fichier (appelé gmon.out) dans le répertoire temporaire d'exécution.

Pour conserver le précieux fichier gmon.out, on ajoute un champ dans le profil astk de type "nom" et dont la valeur est ".../gmon.out". On récupérera alors un fichier de nom gmon.out dans le chemin indiqué.

1.5 Exploitation des résultats

Le fichier produit, sans sur-coût, par l'exécutable instrumenté n'est cependant pas utilisable directement. Il faut exécuter gprof pour le rendre lisible (notez qu'il est nécessaire d'indiquer le nom de l'exécutable instrumenté) :

```
gprof $HOME/dev/codeaster/install/prof/bin/aster gmon.out > listing
```

L'analyse consiste donc à dépouiller le fichier listing produit.

Attention

Il ne faut pas se décourager. Pour une exécution de 30s, on a déjà vu, la commande gprof consommer plus de 5 minutes de CPU. Le temps de gprof ne dépend pas trop (a priori) du temps de l'exécution profilée.

L'interprétation du fichier obtenu (listing) est décrite ci-dessous. Un excellent document décrivant tout le processus de profiling est celui écrit par Jay Fenlason et Richard Stallman : "Gnu gprof The GNU profiler". On le trouve facilement sur le Web.

Remarque

Même si l'on recompile tous les sources d'Aster, la "profondeur" de l'analyse des performances s'arrêtent aux bibliothèques que l'on utilise à l'édition des liens et qui n'ont pas été compilées avec "-pg" . C'est par exemple le cas des routines blas. Le temps consommé dans ces bibliothèques ne peut pas être rattaché aux routines de Code_Aster qui les appellent. Ce défaut peut être important, par exemple, si on veut mesurer les performances des solveurs MUMPS ou MULT_FRONT car une grande partie du temps consommé l'est dans des routines blas.

1.6 Dépouiller les résultats du profiling

Par défaut, le fichier est lourd. Il est possible de limiter l'affichage des infos en jouant avec les options de gprof. Les "temps systèmes" sont indiqués sous forme de nombre d'instructions utilisées.
On va détailler un peu, en commençant par la fin du fichier :

```
Index by function name
 [401] PyArg_Parse            [591] cftabl_   [1000] proc_at_0x1213acb50
 [212] PyArg_ParseTuple       [84] cftyli_    [660]  proc_at_0x1213ad470
[1137] PyArg_ParseTupleAnd    [310] cgmacy_   [453]  proc_at_0x1213ad560
[1605] PyBuffer_FromObject    [79] charme_    [680]  proc_at_0x1213aeac0
[1256] PyCFunction_Fini       [476] chlici_   [1221] proc_at_0x1213aedc0
 [531] PyCFunction_New        [190] chloet_   [217]  proc_at_0x1213b18e0
[1549] PyCObject_AsVoidPtr    [226] chmano_   [629] proc_at_0x1213b1e00Y
```

Chaque fonction appelée lors de l'exécution est repérée par un numéro entre crochet.

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

Juste au dessus :

```
granularity: instructions; units: inst's; total: 201924201580.70 inst's

 <A>  <B>            <C>           <D>  <E>         <F>         <G>
49.6  100384307222  100384307222  161  623505013   623596299 tldlr8_ [16]
31.0   63144941823  62760634601   506  124032874   124101882 rldlr8_ [17]
```

Ce tableau résume les appels les plus fréquents.

- COLONNE <A> : pourcentage du nombre d'instructions exécutées par cette fonction par rapport au total de l'exécution.
- COLONNE <B> : nombre d'instructions cumulées par cette fonction et celles qui précèdent.
- COLONNE <C> : nombre d'instructions pour cette fonction.
- COLONNE <D> : nombre d'appels a cette fonction
- COLONNE <E> : rapport entre la colonne <B> et la colonne <D> (nombre d'instructions moyen par appel de la fonction)
- COLONNE <F> : nombre moyen d'instructions par appel de la fonction et de ses descendants.
- COLONNE <G> : nom de la fonction et son numéro de référence (entre crochets).

Dans cet exemple, la fonction `tldlr8` a pris 49.4% du total du calcul en étant appelée 161 fois.

Enfin, au début du fichier, nous avons l'arbre d'appel complet. Il sera trié par ordre d'appel (on commence par le main et on descend) ou par une fonction (voir les options de `gprof`).

Prenons l'exemple de `tldlr8` :

```
<A>     <B>  <C>              <D>          <E>          <F>

             100263313681.76  14679301.29  161/161      tldlgg_ [15]
[16]    49.7 100263313681.76  14679301.29  161          tldlr8_ [16]
             3129121.03       6207534.02   4485/30537   __upcUpcall [352]
             35974.59         2749927.50   522/195235   jelibe_ [65]
             192341.36        1770419.18   1005/775659  jeveuo_ [56]
             47302.73         140745.02    161/202579   jedema_ [102]
             18938.92         126525.05    322/63148    jeexin_ [196]
             27722.26         85430.33     94/49118     jeecra_ [154]
             17033.41         67779.29     94/13206     jecreo_ [257]
             45068.75         84.88        1044/1075446 jexnum_ [163]
             13618.68         2023.63      161/202581   jemarq_ [205]
             1710.66          0.00         161/3481     infniv_ [853]
```

On repère l'instruction de l'arbre d'appel par le numéro entre crochets à gauche. Ici, le numéro [16] indique la fonction `tldlr8_` (comme indiqué à la fin du fichier par exemple). C'est la fonction-référence (le nœud de l'arbre). Les lignes au dessus sont les appelants de cette fonction (ce sont les fonctions-parents), ceux en dessous sont les fonctions appelées (ce sont les fonctions-enfants). Chaque fonction a deux chiffres principaux : le nombre d'instructions exécutées dans elle-même (instruction " terminale " du `FORTRAN`) et le nombre d'instructions exécutées dans les fonctions-enfants.

```
Fonction-parent
Fonction-parent
...
Fonction-référence
Fonction-enfant
Fonction-enfant
...
```


Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

Pour la fonction-référence :

- COLONNE <A> : numéro de repérage de la fonction-référence.
- COLONNE <B> : le chiffre 49.7 est le pourcentage du nombre d'instructions exécutées par cette fonction-référence par rapport au total de l'exécution (idem tableau précédent)
- COLONNE <C> : nombre d'instructions pour la fonction-référence elle-même.
- COLONNE <D> : nombre d'instructions pour les fonctions-enfants de la fonction-référence.
- COLONNE <E> : nombre de fois ou la fonction a été appelée
- COLONNE <F> : nom de la fonction-référence

Pour les fonctions-parents et les fonctions-enfants :

- COLONNE <A> : vide
- COLONNE <B> : vide
- COLONNE <C> : nombre d'instructions pour la fonction elle-même.
- COLONNE <D> : nombre d'instructions pour les descendants de la fonction
- COLONNE <E> : donne deux chiffres a/b dont le sens varie suivant le type de fonction (parent ou enfant par rapport à la fonction référence) :
  - Pour les fonctions-parents (au-dessus de la fonction référence) a/b : <a> est le nombre de fois où la fonction-référence a été appelée par cette fonction-parent par rapport au nombre total <b> d'appels de la fonction-référence.
  - Pour les fonctions-enfants (en-dessous de la fonction référence) a/b : <a> est le nombre de fois où la fonction-enfant a été appelée par la fonction-référence par rapport au nombre total <b> d'appels de la fonction-enfant.
- COLONNE <F> : nom de la fonction

Remarques :

Si le nombre d'instructions pour les descendants d'une fonction vaut zéro, c'est que la fonction considérée n'en appelle aucune autre. On est " au bout " de l'arbre, il n'y a que des appels FORTRAN de base dans la fonction (c'est le cas de `infniv` par exemple).
Pour une fonction-référence donnée, si on fait la somme des <a> dans la colonne <E> des fonctions parents, on obtient le nombre d'appels total de la fonction référence.
Pour une fonction-référence donnée, si on fait la somme des colonnes <C> et <D> de ses fonctions-enfants, on obtient le chiffre de la colonne <D> de la fonction-référence.

1.7 Analyse de l'exemple

Dans l'exemple présenté, la fonction tldlr8 est coûteuse puisqu'à elle-seule, elle représente près de la moitié du nombre d'instructions total de l'exécution. On voit également que ce sont ses propres instructions qui prennent du temps et non l'appel à ses fonctions-enfants (le rapport entre les deux atteint 1000). Comme seule la fonction `tldlgg` appelle `tldlr8`, il faut regarder l'arbre d'appel pour cette fonction. On voit alors que c'est l'algorithme de contact/frottement (`fropgd`) qui est le plus glouton (les 2/3 des appels à `tldlgg` sont faits par l'algorithme de contact).

1.8 Génération d'un graphe avec *gprof2dot*

1.8.1 Présentation

Pour faciliter l'interprétation d'un *profiling*, on décrit dans cette section un petit utilitaire Python (*gprof2dot*) qui transforme le fichier texte produit par `gprof` en un graphe plus simple à lire.

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

Le graphe produit est celui des routines parcourues avec report des colonnes <B> et <C> (en % du nombre total d'instructions), du nombre d'appels de la routine. Par ailleurs les cellules du graphe sont coloriées (du bleu vers le rouge) pour identifier très vite les chemins critiques.
On trouvera plus d'informations sur la page web du développeur de *gprof2dot* : http://code.google.com/p/jrfonseca/wiki/Gprof2Dot.

1.8.2 Utilisation

Une fois le fichier d'*output* produit par `gprof` récupéré, celui-ci s'appelant `listing`, on exécutera la commande suivante dans un terminal :

```
cat listing | gprof2dot.py | dot -Tpng -o graphe.png
```

Un exemple du type d 'image produite est donné dans la figure suivante.

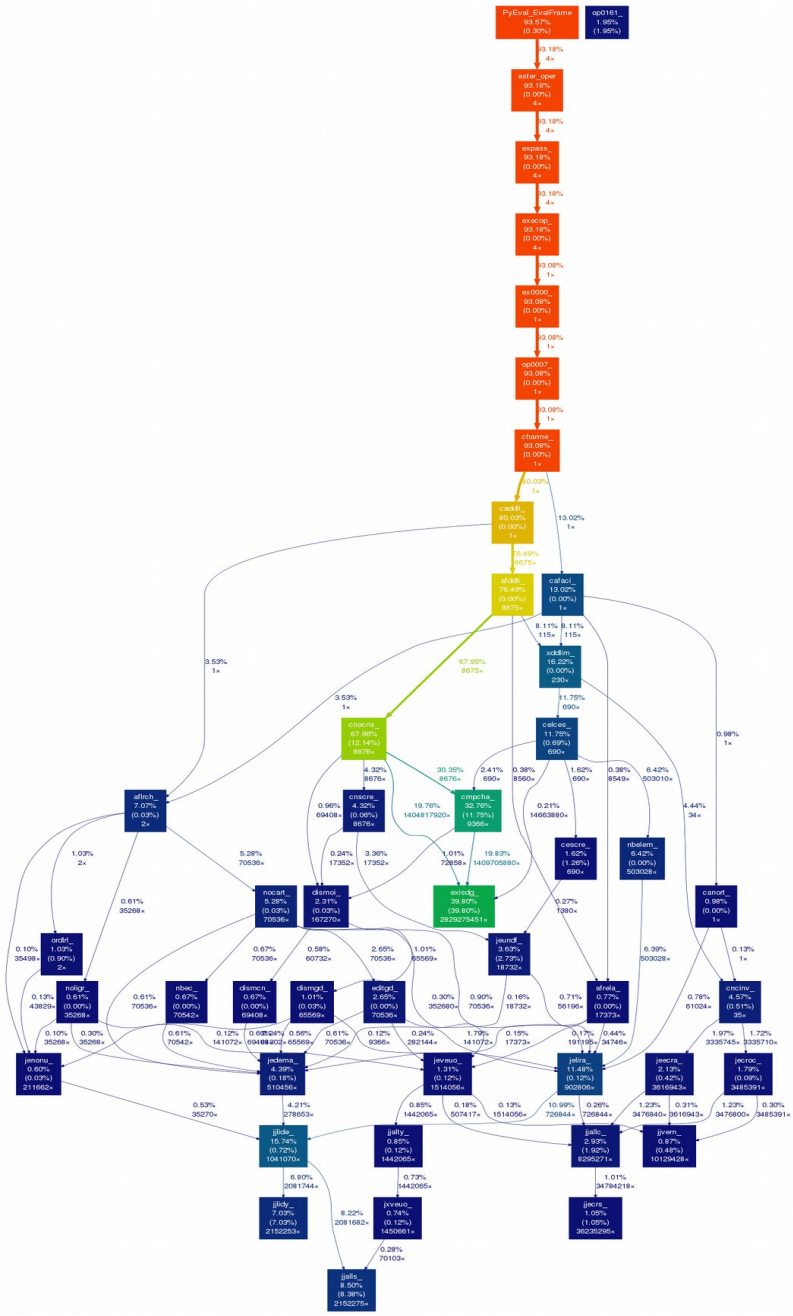


Figure 1 : graphe d'appel généré avec *gprof2dot*

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)

Document diffusé sous licence GNU FDL (http://www.gnu.org/copyleft/fdl.html)