Manuel de Descriptif Informatique
Fascicule D1.05 : -
Document D1.05.01

Mesurer les performances (CPU) sur l'AlphaServer ou sur Linux

Résumé :

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

Sur l'AlphaServer, ces outils ne nécessitent pas une recompilation des sources Aster. On utilise pour cela l'outil `atom`. L'inconvénient de cet outil (spécifique alphaserver) est que l'instrumentation de l'exécutable entraîne des sur-coûts d'exécution qui peuvent être très importants (jusqu'à 10 fois le coût initial). Dans ces conditions, il est difficile d'être sûr de la pertinence de la mesure.

Sur Linux, on utilise la méthode traditionnelle : on recompile tous les sources avec l'option "`-pg`" et on utilise l'outil `gprof`. Le sur-coût de l'instrumentation est négligeable.

1 Sur Alphaserver

1.1 Préparer le profiling

On travaille à partir de l'exécutable que l'on compte utiliser pour lancer son étude Aster :

- Exécutable natif : Recopier l'exécutable aster sur un répertoire local qui vous appartient sur le serveur (l'exécutable natif est dans `/aster/v7/NEW7/` sur le serveur et se nomme `asterd` ou `asteru` en mode debug ou non).
- Exécutable privé : Préparer votre surcharge comme d'ordinaire avec ASTK ou run_aster et construisez votre exécutable.

Il faut ensuite modifier l'exécutable a l'aide de l'outil `atom`.

Sur votre répertoire contenant l'exécutable Aster que vous voulez profiler :
```
atom -tool hiprof votre_executable
```

Le programme va créer un nouvel exécutable nommé `<votre_executable.hiprof>`

1.2 Faire le profiling

A l'aide d'ASTK ou `run_aster` il faut utiliser le nouvel exécutable en surcharge. Il faut impérativement modifier le script de lancement d'Aster car lors de l'exécution en profiling, le fichier `<votre_executable.hiout>` sera créé dans le répertoire temporaire de calcul. Il faut donc le copier dans le répertoire adéquat.
Pour ASTK :

1) Préparez l'étude (fichier, surcharges des catalogues, nouvel exécutable "profilé", bases, temps, mémoire et options diverses).
2) Ajoutez le script `btc` en RESULTAT dans l'onglet SURCHARGE.
3) Lancez le calcul. Le calcul ne sera pas exécuté (une boite de dialogue vous en avertit) mais le script (`btc`) sera créé.
4) Modifiez le script `btc` en l'éditant et en rajoutant la ligne suivante à la fin :
```
cp votre_executable.hiout /chez_vous/votre_executable.hiout
```

Prenez garde ! Pour toute modification dans le profil d'exécution (en particulier temps et mémoire), il est indispensable de recréer le `btc` et de le modifier.
Apres exécution, on se retrouve avec deux fichiers :
```
votre_executable.hiout
votre_executable.hiprof
```

Ces deux fichiers doivent être dans le même répertoire. On exécute alors `gprof` en redirigeant la sortie standard:
```
gprof votre_executable votre_executable.hiout > ResultatProfil
```

Vous avez désormais un fichier `<ResultatProfil>` qui est le résultat de l'analyse.

Pour les options possibles, faire un `man gprof`. Quelques options utiles :

`gprof -a`
Evite l'affichage des fonctions statiques, en particulier les appels systèmes qui alourdissent le fichier

`gprof -a -f jeveuo_`
Limite l'affichage à la fonction designee

Attention :

> *Pour une routine FORTRAN, ajoutez impérativement un _ (underscore) à la fin du nom de la routine et enlevez l'extension .f*

1.3 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.

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 163144941823   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
Fonction-enfant
Fonction-enfant
…

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.*

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).

2 Sur Linux

2.1 Instrumentation avec `f77 -pg` (ou `cc -pg`)

Sur la machine Linux/Rocks clpaster (cluster de PC du département AMA), le problème de la recompilation intégrale des sources est moins crucial que sur l'alphaserver : on peut recompiler entièrement Aster en moins de 30 minutes "elapse".

Pour réaliser cette recompilation avec Astk, il faut :

- "surcharger" tous les sources (F77 et C). Pour gagner du temps, on peut concaténer les sources F77 en "paquets" (300 routines par exemple).
- modifier le fichier "`config.txt`" pour ajouter l'option "`-pg`" sur les 5 lignes suivantes :
  - `OPTL | f90 | ? | -v -pg`
  - `OPTC_D | cc | ? | -c -g -pg -DP_LINUX`
  - `OPTC_O | cc | ? | -c -pg -DP_LINUX`
  - `OPTF_D | f90 | ? | -c -g -pg -I/opt/mpich2-1.0.1/include`
  - `OPTF_O | f90 | ? | -c -O2 -pg -I/opt/mpich2-1.0.1/include`

Le fichier `config.txt` ainsi modifié permet d'instrumenter le code en mode "`debug`" et "`nodebug`". Le mode "`nodebug`" 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.

J'ai 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.

A titre d'exemple, j'ai profilé le test `ssnv506c` et j'ai 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.

2.2 Exécution du Code instrumenté avec `Astk`

Une fois cette instrumentation faite, il faut exécuter l'étude que l'on veut "profiler" avec l'exécutable que l'on vient de produire. Le problème est que l'exécution de l'étude produit un fichier (appelé `gmon.out`) dans le répertoire temporaire d'exécution. Ce fichier est donc perdu en fin d'exécution si on ne prend pas de précautions.

Pour conserver le précieux fichier `gmon.out`, il faut utiliser `Astk` en interactif et cliquer le bouton "`lancer pre`" (au lieu du classique "`lancer run`"). Cette option d'Astk permet de préparer l'environnement d'exécution. On se place ensuite dans le répertoire préparé et on "lance" Aster manuellement. Il s'agit de la même "astuce" que pour l'utilisation d'un débuggueur.

2.3 Exploitation des résultats

Une fois l'étude exécutée et le fichier "gmon.out" récupéré, on peut analyser ce fichier avec la commande :

```
gprof mon_executable gmon.out > listing
```

L'interprétation du fichier obtenu (listing) est la même que celle décrite au [§1.3]. 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 d'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.