Profilage, debogage, optimisation

Romaric DAVID

École d'Automne Informatique Scientifique, 2 Décembre 2008

Plan

Plan

- Profiling
- Débogage
- Optimisations de programmes

Introduction

Objectifs du cours

Ce cours vous présentera les outils d'analyse de vos programmes. En particulier, vous pourrez :

- Traquer des bugs dans les codes
- Déterminer les parties du code gourmandes en temps de calcul
- Découvrir quelques pistes permettant d'optimiser le programme

1 Débogage

Dans cette section, nous présenterons les principes des outils de débogage et les points importants dans leur utilisation. Déboguer un programme consiste à chercher pendant son exécution l'origine d'une erreur de comportement. Une erreur de comportement d'un programme lié à un accès mémoire frauduleux (hors des zones attribuées au programme) produit le fameux message "sementation fault (core dumped)". Cela signifie que l'image mémoire du processus a été placée dans un fichier appelé `core`. Ce fichier peut servir de paramètre d'entrée à toutes les manipulations présentées ci-dessous.

Pourquoi déboguer ?

- Programme produisant un résultat incorrect
- Un plantage du programme

- Un problème lié à la gestion de la mémoire [1] ⇒ segmentation fault (core dumped)

Le principe fondamental du débogage de programme est de faire un suivi pas à pas (instruction par instruction) de son exécution. Le programme qui permet de faire ceci s'appelle un débogueur (version mal francisée de debugger). À partir des indications dans le code compilé, le débogueur peut faire le lien entre le code source et le code assembleur produit.

Comme il y a un grand nombre d'instructions dans un programme, le débogueur permet aussi de fixer des **points d'arrêt**. À chaque fois que le programme passe par ce point, il s'arrête et le débogueur reprend la main.

Pour examiner les valeurs, le débogueur permet également de surveiller des variables. Ceci permet de savoir à quel étape de l'exécution d'un programme une variable prend une valeur incohérente.

Afin de permettra au débogueur de faire le lien entre code assembleur et code source, il faut compiler le programme avec l'option `-g`. Cette option désactive par défaut toute optimisation. Il est néanmoins possible de combiner optimisation et présence de symboles de débogage. Les symboles de débogage associent les structures de contrôle du langage au code assembleur. S'ils sont présents (associés au ficher source), le débogueur affichera le code source correspondant lorsque le programme est arrêté.

Au niveau système, les débogueurs (testé avec idb, gcc, totalview 7) utilisent l'appel système ptrace[2]. Cette appel permet de contrôler l'exécution d'un autre processus.

Les débogueurs disposant de possibilités étendues permettent d'afficher les données, également sous forme graphique. Un autre gros travail de mise en forme consiste en l'affichage de données structurées ou objet. C'est un intérêt majeur des débogueurs (p/r à l'affichage manuel). Les débogueurs sont souvent construits sous la forme d'un outil textuel, auquel peut s'adjoindre une interface graphique.

Débogueurs en mode texte

- dbx (Solaris, IRIX, AIX)
- gdb (tous)
- idb (Intel, Linux IA64 et X86-64)
- tv8cli (Totalview)
- pdb (pour python)

Des interfaces graphiques les pilotent :

- ddd (Data Display Debugger). Interfaçable avec dbx, gdb
- DDT (Allinea)
- Interface de totalview
- Kdbg (gdb)
- Eclipse Ptp (Parallel tools platform)

[1] Classe d'outils à part

[2] Cf `http://falco.next-touch.com/repository/securite_informatique/linux/anti-anti-ptrace.pdf`

2 Profiling

Une question que l'on se pose souvent quand on attend qu'un programme se termine est de savoir quelle fonction représente la plus grande partie du temps d'exécution. Pour **mesurer** le temps passé dans une fonction, on peut utiliser la méthode simple suivante.

Une méthode simple de mesure du temps passé

```
time=get_time(now)
fonction_calcul(...)
time_spent=get_time(now)-time
save(time_spent)
```

Notez que la sauvegarde du temps passée peut être compliquée (gestion variable globale, ...)

Limite de la méthode :

- Facile pour une fonction, mais pour plusieurs ?
- Facile pour une métrique, mais ...

⇒ nécessité d'automatiser.

Le but de du profiling (analyse comportementale) est de connaître le comportement du programme à l'exécution. En premier lieu, les profilers peuvent s'intéresser au programme tout seul (sans examiner l'environnement dans lequel il s'exécute), c'est-à-dire déterminer le temps passé dans différentes fonctions (outils comme gprof par exemple).

Ainsi, sans avoir d'idée à priori sur la fonction coûteuse (ce qui peut être le cas quand on n'est pas le développeur du code), le profiler peut pointer du doigt une fonction particulière. Les profilers renvoient en général le temps passé dans une fonction ainsi que le nombre d'appels. Si une fonction n'est pas appelée assez souvent, cela peut rendre compte de bugs dans son algorithme (fonction appelée dans les mauvaises conditions, ...).

Des outils plus évolués permettent d'analyser l'interaction du programme avec son environnement. Par exemple, un *profiler* peut indiquer le nombre de défauts de cache survenus durant l'exécution d'un programme.

La séquence de profiling commence par une phase d'instrumentation (= **ajout d'instructions**) destinées à rassembler l'information. Après l'exécution du programme modifié, les données issues de l'instrumentation sont rassemblées et analysées par un outil spécifique.

Données recueillies par le profiler

- Temps d'exécution de chaque fonction (profil plat)
- Métriques utilisateur
- Callers/Callees : représentation du graphe d'appel du programme
- Temps inclusive : fonction appelante + fonction appelée
- Exclusive : fonction appelante uniquement

L'instrumentation peut être statique. Elle sera dans ce cas effectuée au plus tard à la compilation. L'instrumentation dynamique a lieu à l'exécution et fait appel à des techniques plus complexes.

2.1 Instrumentation statique / source

2.1.1 Gprof

Des instructions de profiling sont ajoutées à la compilation. Le rôle peût être confié au compilateur et/ou a l'éditeur de liens. Quand le rôle est confié au compilateur, c'est souvent en vue de l'analyse avec l'utilitaire gprof. Pour chacune des fonctions du programme, le compilateur ajoutera les instructions permettant de connaître temps passé et nombre d'appels.

Utilisation du compilateur avec gprof

Les compilateurs usuels permettent d'automatiser la mise en place des instructions de mesure analysables par gprof.

- Compilateur Intel : -p
- Gnu : -pg
- -g ⇒ affichage ligne par des informations (utilisation gprof -l)

Comment cela fonctionne-t'il ? **Chaque fonction** du programme est modifiée à la compilation de telle sorte à mettre à jour les structures de données stockant la fonction appelante et le nombre d'appels. Le temps d'exécution de la fonction est déterminé par échantillonage.

De plus, le démarrage et l'arrêt du programme sont modifiés de telle sorte à allouer de la mémoire pour les structures de données nécessaires et prévoir la sauvegarde des informations à la fin du programme. Un gestionnaire de signal d'horloge est également ajouté pour stocker les informations au fur et à mesure. Pendant l'exécution du programme, le compteur de programme est examiné environ 100 fois par seconde (dépendant du système). Un histogramme est généré à partir de ces informations. Seul le temps passé pendant que le programme est en exécution (càd pas en attente d'entrée/sorties) est pris en compte.

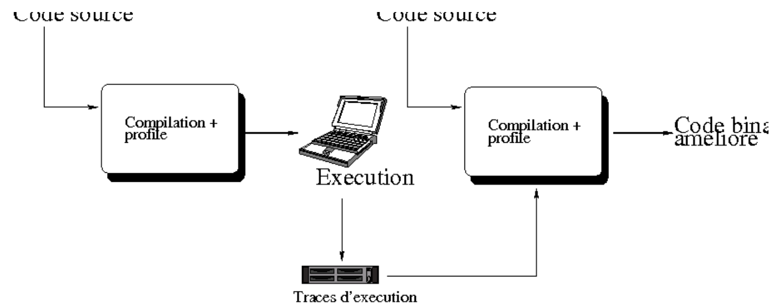
Que se passe-t-il pour les fonctions non instrumentées (bibliothèques externes) par exemple ? Le profiling fonctionne, mais la seule information relative à ces fonctions est le temps passé. Le nombre d'appels, information nécessitant une re-compilation, n'est pas disponible. Cela rend le graphe d'appels incomplet et moins facile à utiliser.

2.2 Optimisation assistée par le profil d'exécution

Certains compilateurs comme le compilateur Intel ou Portland permettent de prendre à compte à la compilation d'informations de profiling obtenues lors d'une précédente exécution. Cela peut indiquer au compilateur le comportement typique d'un programme et ainsi lui indiquer sur quelles régions ou sur quels branchements particulièrement optimiser le code. Durant l'optimisation interprocédurale, cela permet aussi de déterminer les enchaînements de routines.

Optimisation assistée

Le schéma d'exécution est le suivant et vise a réinjecter dans une deuxième compilation les informations de profiling issues d'une exécution.



Les gains à attendre de cette technique sont fonction du programme.

2.2.1 gcov / Outils équivalents

Ggov[3] permet de connaître le pourcentage de temps d'exécution passé dans chaque ligne du code (test de couverture). Il peut se prêter au profiling ligne par ligne d'un code source, **uniquement avec le compilateur gcc**.

Utilisation de gcov

Pour analyser ligne par ligne le déroulement d'un programme :

- Compilation avec `gcc -fprofile-arcs -ftest-coverage` ⇒ génération d'un graphe de flôt
- Exécution du programme
- Après l'exécution, `gcov source.c`
- Produit `source.c.gcov`, Affiche le % de ligne exécutées
- Analyse de `source.c.gcov` par ggcov ou ... manuelle

Permet aussi de savoir si certaines lignes ne sont jamais exécutées

Outils équivalents dans la famille Intel

On retrouve le schéma CEA (compilation, exécution, analyse). Options/outils à utiliser :

- Compilation avec `-prof-genx -prof-dir/usr/profiled`
- Fusion des différents fichiers obtenus : `profmerge`
- Extraction et présentation sour forme Html : `codecov`

2.3 Tau

Retour sur gprof

La caractéristique des outils comme gprof est de faire analyser *après l'exécution* du programme un *fichier de traces* généré pendant son exécution. Le fichier de traces de gprof est spécifique aux quelques informations que gprof sait traiter. Cette séquence récupération de traces suivie de l'analyse se retrouve dans plusieurs outils.

⇒Nécessité d'enrichir le contenu des traces.

[3] `http://gcc.gnu.org/onlinedocs/gcc/Gcov-Intro.html`

Plus d'événements

Les outils permettant de stocker des événements définis par l'utilisateur utilisent un format de fichier de traces qui leur est propre, même si une tentative de normalisation existe : Open Trace Format[4].

Les outils utilisant ce genre de traces sont initialement des outils dédiés à MPI. On trouve par exemple les paires Intel Trace Collector /Intel Trace Analyzer, Vampir Trace / Vampir et MPE / jumpshot .

Tau[5] est un exemple intéressant d'outil plus générique, même si toujours tourné vers le parallélisme. Par rapport à gprof, il peut fonctionner avec ou sans pré-compilation.

Pour utiliser tau, il faut compiler (`./configure ; make; make install`) l'outil en choisissant les grandeurs que l'on veut instrumenter. Il est conseillé de compiler une bibliothèque par grandeur. Cela conduira à la création de bibliothèques dynamiques à lier avec son exécutable par le biais de *w*rapper scripts permettant de compiler son exécutable en le liant avec les bibliothèques dynamiques Tau. Certaines de ces bibliothèques (en particulier celles liées à MPI) peuvent également être utilisées avec l'exécutable sans recompilation préalable (bibliothèques d'interposition). Les bibliothèques Tau (une par fonctionnalité) produisent des fichiers de trace au format Otf (et dans d'autres formats également). Mais que sait faire tau ?

- Profiler classiquement le programme
- Profiler mpi
- Profiler les appels de fonctions
- Regarder l'utilisation de la mémoire

À retenir

- Orienté programmes parallèles
- Couplage compilateur / profiler
- Re-compilation nécessaire dans certains cas
- Couplage possible avec optimisations du compilateur (gprof)

3 Instrumentation dynamique / binaire

Les outils vus dans cette partie du cours nécessitent une re-compilation du programme, ce qui implique bien évidemment d'en avoir les sources. Il existe des outils permettant d'analyser des programmes sans re-compilation, par des techniques analysant à la volée son exécution. On parle alors d'instrumentation dynamique.

L'instrumentation dynamique permet d'analyser pendant l'exécution le comportement d'un programme. Plusieurs métriques peuvent être rassemblées à la volée. Il existe plusieurs techniques pour capturer à la volée les informations nécessaires :

[4] `http://www.paratools.com/otf.php`
[5] `http://www.cs.uoregon.edu/research/tau/`

- Bibliothèques d'interposition : les bibliothèques **dynamiques** utilisées par votre application sont remplacées par celles de l'outil d'analyse. valgrind, Tau dans certains modes de fonctionnement.

- Exécution du code sur un émulateur du système ou machine virtuelle (un pseudo-système instrumenté de partout) : valgrind, pin

Nous présenterons Valgrind et Pin dans la suite.

3.1 Valgrind

Valgrind est une machine virtuelle qui pratique la *t*raduction binaire : traduction à partir de la forme compilée du programme vers une représentation intermédiaire. Cette représentation est ensuite analysée pour y insérer les instructions de profiling demandées. Ensuite, elle est exécutée sur le processeur virtuel. Le temps d'exécution est multiplié par 4 à 50. par l'intermédiaire de la machine virtuelle. Ce fonctionnement explique la nette dégradation des performances. Valgrind a été conçu initialement pour le débogage des allocations mémoire quelque soit le langage utilisé[6].

Valgrind

- d'analyser le comportement du cache (cachegrind)

- d'analyser (dynamiquement) les graphes d'appel (callgrind)

- d'analyser les accès aux données par des threads concurrents (helgrind) ⇒ on s'approche du debogage

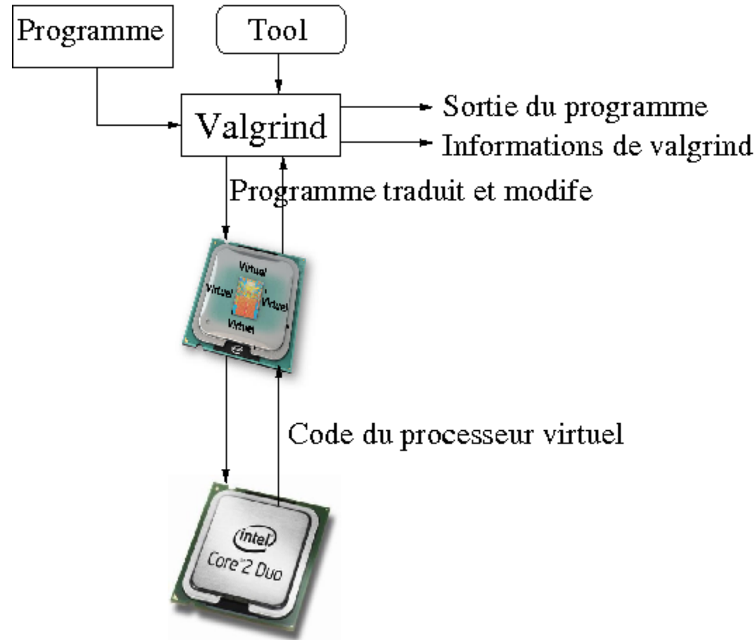
Pour l'analyse des accès mémoire, on peut considérer Valgrind comme un débogueur. C'est d'ailleurs l'exemple le plus classique d'utilisation de valgrind. Nous regardons ici la sortie de l'exécution de la commande `uname -a`· sous valgrind :

```
david@angdmath1:~$ valgrind --tool=memcheck /bin/uname -a
==11776== Memcheck, a memory error detector.
==11776== Copyright (C) 2002-2007, and GNU GPL'd, by Julian Seward et al.
==11776== Using LibVEX rev 1854, a library for dynamic binary translation.
==11776== Copyright (C) 2004-2007, and GNU GPL'd, by OpenWorks LLP.
==11776== Using valgrind-3.3.1-Debian, a dynamic binary instrumentation fra
k.
==11776== Copyright (C) 2000-2007, and GNU GPL'd, by Julian Seward et al.
==11776== For more details, rerun with: -v
==11776==
Linux angdmath1 2.6.26-1-amd64 #1 SMP Sat Nov 8 18:25:23 UTC 2008 x86_64 GN
ux
==11776==
==11776== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 8 from 1)
==11776== malloc/free: in use at exit: 0 bytes in 0 blocks.
==11776== malloc/free: 30 allocs, 30 frees, 3,673 bytes allocated.
```

[6]a la différence de Electric Fence, destiné uniquement au `malloc` de C

```
==11776== For counts of detected errors, rerun with: -v
==11776== All heap blocks were freed -- no leaks are possible.
```

Valgrind - en images



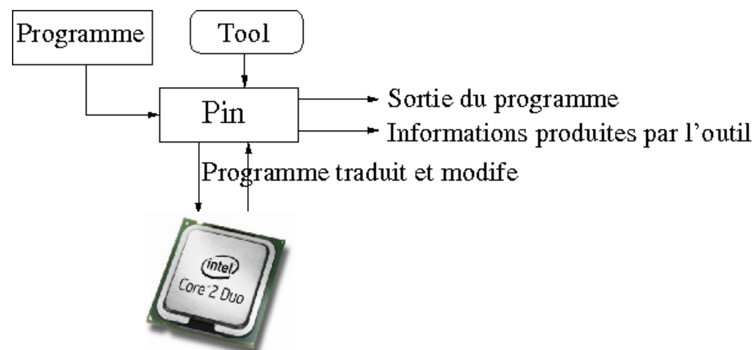
3.2 Pin

Pin est un moteur d'exécution provenant d'Intel permettant de tracer et d'analyser les performance d'un programme à un niveau très fin (jusqu'à instruction par instruction). L'instrumentation se fait à l'aide de *p*intools (à écrire en C++ et à compiler soi-même). Ces pintools utilisent l'API pin pour déterminer quel est le type de l'instruction et effectuer l'action correspondante.

Avant d'exécuter une instruction, pin vérifie si elle doit donner lieu à analyse. Si oui, elle est ré-écrite avant d'être exécutée par le processeur (pas de machine virtuelle). Sinon, elle est exécutée sans modification par le processeur.

Pin permet également de s'attacher / détacher d'un programme et d'y remplacer à la volée une routine par une autre. Pin entraîne ralentissement, dépendant du pintool, toujours supérieur à 5% d'après les auteurs.

Pin - en images



4 Optimisation

Nous présentons ici les grandes stratégies d'optimisation à garder en tête lors de l'écriture d'un code.

4.1 Don't do it yourself

Ai-je vraiment besoin d'écrire du code ?

Il y a de grandes chances que les opérations numériques de base dont vous avez besoin aient déjà été écrites par quelqu'un d'autre. Quelques sources à regarder :

- Netlib : `http://www.netlib.org/`[7] Regroupement de bibliothèques mathématiques et d'articles
- ACM Calgo : `http://oldwww.acm.org/pubs/calgo/`. Logiciels correspondant aux articles parus dans Transactions on Mathematical Software et d'autres revues
- `http://en.wikipedia.org/wiki/List_of_numerical_analysis_software` fournit de nombreux liens sur des logiciels d'analyse numérique

L'intérêt des bibliothèques déjà faites est que les plus diffusées d'entre elles disposent d'une API reprise dans les versions optimisées fournies par les éditeurs de logiciels.

Votre objectif peut être de produire des nouvelles bibliothèques performantes. Pour cela, voici quelques pistes d'optimisation. Hors accès disque, il y a deux grandeurs coûteuses dans le temps d'exécution d'un programme : le temps passé à calculer et le temps passé à échanger des données avec la mémoire. Il faut donc bien évidemment réduire ces deux grandeurs afin d'accélérer son programme. Rappelez-vous que la seule grandeur sensible à l'utilisateur est le temps d'exécution.

Mais je veux vraiment écrire du code !

Commençons par réfléchir à la diminution du temps de calcul. Voici quelques pistes.

- Ai-je vraiment besoin de tous ces calculs ? $\Rightarrow$ penser pré-calculs, ré-utilisation des résultats, calcul sur des zéros
- Est-ce que j'utilise le bon algorithme ? Quelle est sa complexité (nombre d'opérations) ? En existe-t'il d'autres moins coûteux ?
- Que me coûte l'accès aux données ? $\Rightarrow$ Penser structures de données
- Est-ce que mon algorithme est parallélisable ?

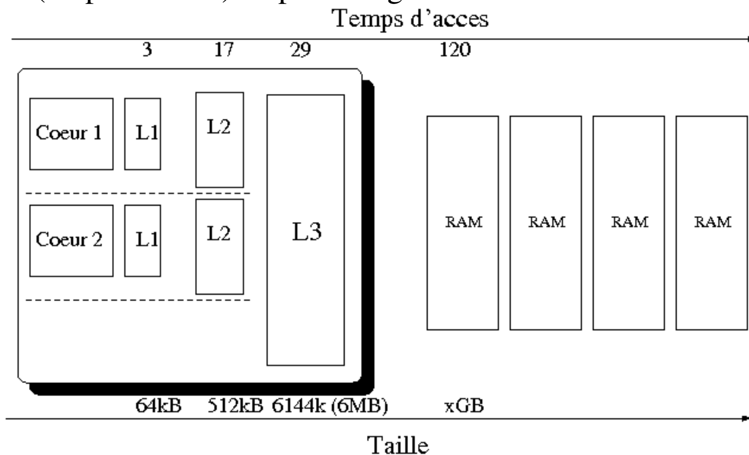
Cependant, le nombre de coeurs par processeur augmentant, les gigaflops étant de moins en moins coûteux (cartes graphiques, Cell, ...), l'accès aux données en mémoire devient le point critique des applications. Il s'agit donc d'une des principales sources d'optimisation.

Rappelons l'architecture des processeurs avec le ratio des coûts d'accès à la mémoire en fonction de son éloignement du processeur.

[7] Version plus lisible: `http://www.netlib.org/master/expanded_liblist.html`

Mais où sont mes données ?

La mémoire d'un ordinateur est dite hiérarchique, avec des niveaux. Nous les examinons du plus proche (du processeur) au plus éloigné[8]



.

Le cache est rempli par le processeur automatiquement : à chaque chargement d'une donnée (un élément de tableau), un morceau de tableau permettant de remplir une ligne de cache est chargé.

La stratégie générale consiste à utilise le plus souvent les données dans le cache. On dit alors qu'on veut profiter de la localité des données. Qu'est-ce que c'est ?

Localité des données

- Localité spatiale : accéder à des données proches en mémoire. Elles seront probablement dans le cache.
- Localité temporelle : réutiliser les données qui sont de toute façon dans le cache

Pour améliorer l'efficacité des accès mémoire, un programme s'efforcera de profiter de ces deux principes de localité. Pour cela, les règles sont classiques. Elles sont d'autant plus importantes que dans les processeurs multi-coeurs, il y a concurrence pour l'accès à la mémoire aggravée par le nombre de coeurs.

Quelques règles

Pour utiliser au mieux les données du cache on peut :

- écrire des algorithmes travaillant par blocs, blocs qui tiennent dans le cache (principe des blas/lapack)
- écrire des programmes qui ne provoquent pas de défaut de cache ("remplacer structures de tableaux par tableaux de structures", fusionner les boucles, inverser les boucles). Certains compilateurs/optimiseurs peuvent le faire.
- . . . écrire des programmes avec un informaticien à côté de soi.

⇒ influence sur la lisibilité et la maintenabilité du programme. Quoiqu'il en soit, il faut augmenter le ratio nombre d'opérations / nombre d'accès mémoire.

[8] `http://www.x86-secret.com/articles/cpu/k8-3/amd64-6.htm`

5 Conclusion

Nous avons vu dans ce cours différents outils pour torturer un code dans tous les sens : voir ce qui ne fonctionne pas, comprendre où sont les points bloquants, écrire des programmes qui sauront tirer parti des architectures multi-coeurs.