Profiling

Romaric DAVID

École d'Automne Informatique Scientifique
3 Décembre 2008

Plan

- Profiling
- Débogage
- Optimisations de programmes

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

Plan

Pourquoi déboguer ?

- Programme produisant un résultat incorrect
- Un plantage du programme
- Un problème lié à la gestion de la mémoire [1] $\Rightarrow$ segmentation fault (core dumped)

[1] Classe d'outils à part

Principe du débogage

- Suivi pas à pas du déroulement du programme
- Lien entre code source et assembleur
- Fonctionnalités importantes
  - Points d'arrêt : interrompre le déroulement du programme à cet endroit
  - Surveillance de variables : réagir aux modifications de variable ("mais où diable cette variable est-elle modifiée ?")
- Compilation avec symboles de débogage `-g`. Peuvent être associés à optimisation.
- Lancement sous le débogueur ou attachement à un programme en cours
- Mise en place points d'arrêt, surveillance de variables
- Examen de valeurs de variables[2]
- Poursuite de l'exécution
- Techniquement : utilisation de l'appel système ptrace

[2] y compris fonctions de visualisation évoluées (2D, 3D, tranches)

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)

Plan

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, . . .)
$\Rightarrow$ nécessité d'automatiser.

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, . . .)
$\Rightarrow$ nécessité d'automatiser.

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 . . .

$\Rightarrow$ nécessité d'automatiser.

Analyse du comportement

Permet de connaître le comportement du programme **à l'exécution**. Permet de **mesurer** :

- Où un programme passe le plus clair de son temps
- Le nombre d'appels à une fonction

Les outils d'analyse de comportement (profilers) permettent de consolider l'ensemble des mesures réalisées.

Analyse du comportement

Permet de connaître le comportement du programme **à l'exécution**. Permet de **mesurer** :

- Où un programme passe le plus clair de son temps $\Rightarrow$ piste d'optimisation
- Le nombre d'appels à une fonction $\Rightarrow$ piste de débogage

Les outils d'analyse de comportement (profilers) permettent de consolider l'ensemble des mesures réalisées.

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

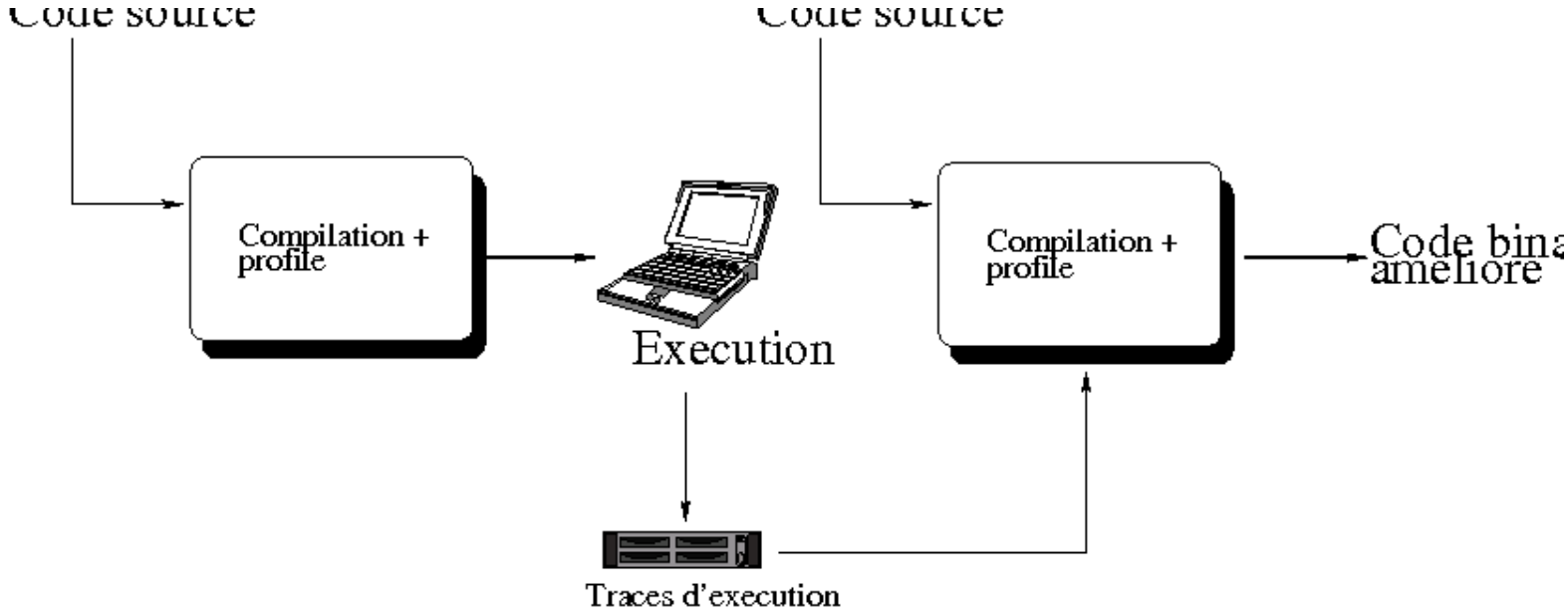
Utilisation du compilateur avec gprof

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

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

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.



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`

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.
$\Rightarrow$Nécessité d'enrichir le contenu des traces.

Plus d'événements

- Outils permettant à l'utilisateur de définir ses propres événements
- Consolidation, agrégation des résultats automatique
- Affichage à l'aide de l'outil
- Format de traces spécifique

À retenir

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

Plan

Valgrind I

Valgrind est une machine virtuelle (processeur synthétique) exécutant le code du programme. Avant d'être exécuté, le code binaire original est interprété (traduction binaire) et éventuellement enrichi des instructions de profiling. Ensuite, il est re-traduit (traduction juste-à-temps) avant d'être exécuté sur le processeur synthétique.
Maintenant étendu à l'aide d'autre *o*utils, Valgrind permet aussi de :

- 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) $\Rightarrow$ on s'approche du debogage

Valgrind II

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 framewor
```

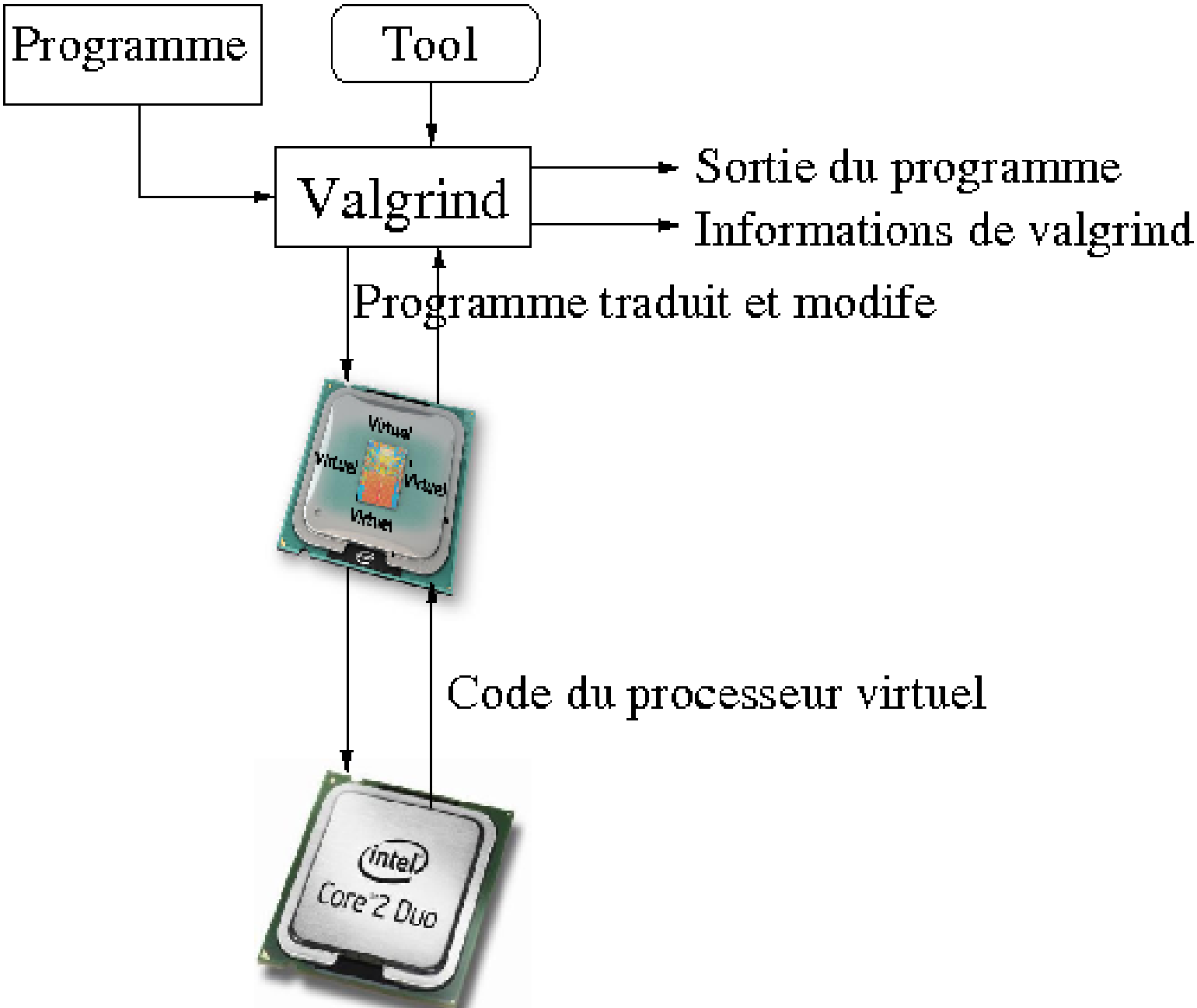
Valgrind III

```
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 GNU/Lin
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.
```

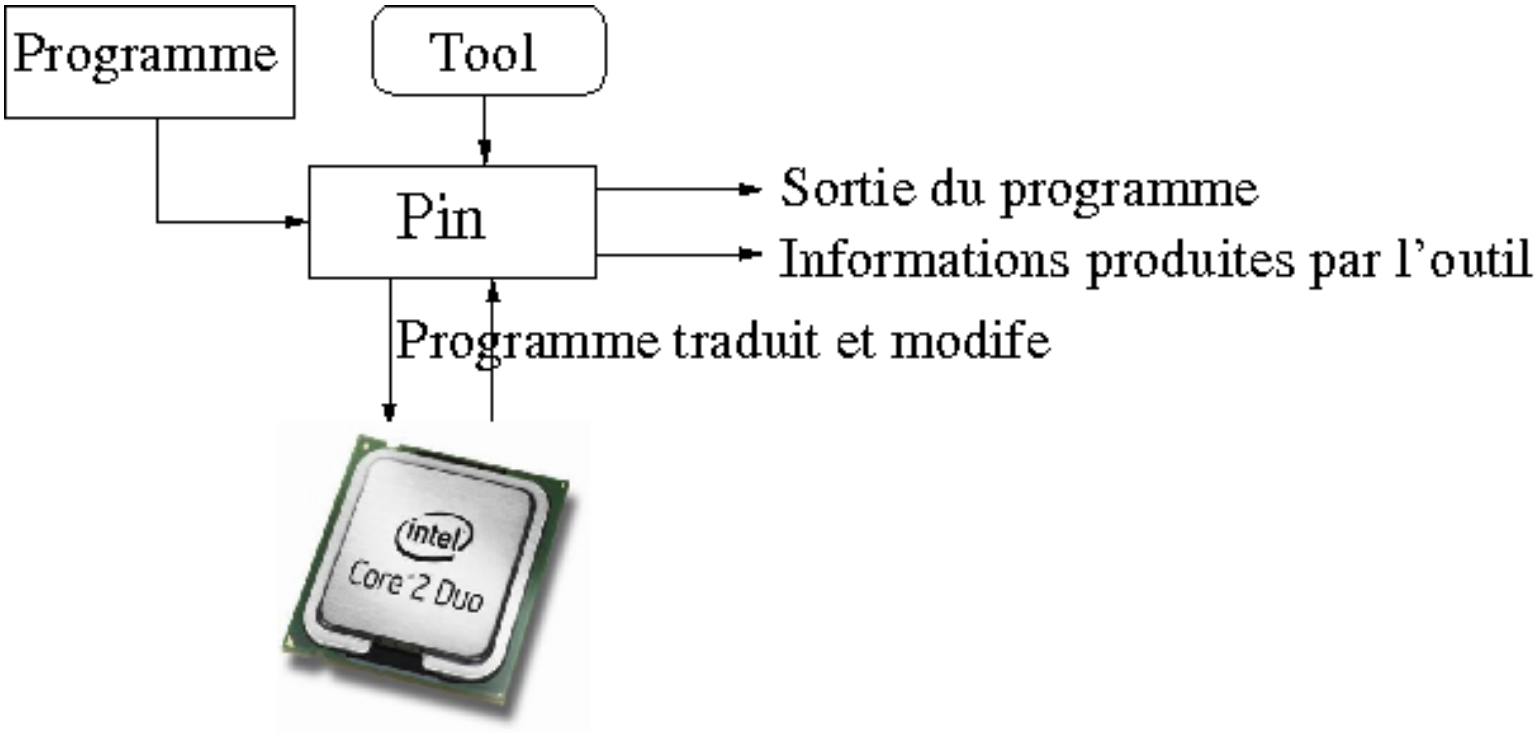
Valgrind IV

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

Valgrind - en images



Pin - en images



Plan

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/`[3]
- ACM Calgo : `http://oldwww.acm.org/pubs/calgo/`.
- `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.

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

Mais je veux vraiment écrire du code ! I

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

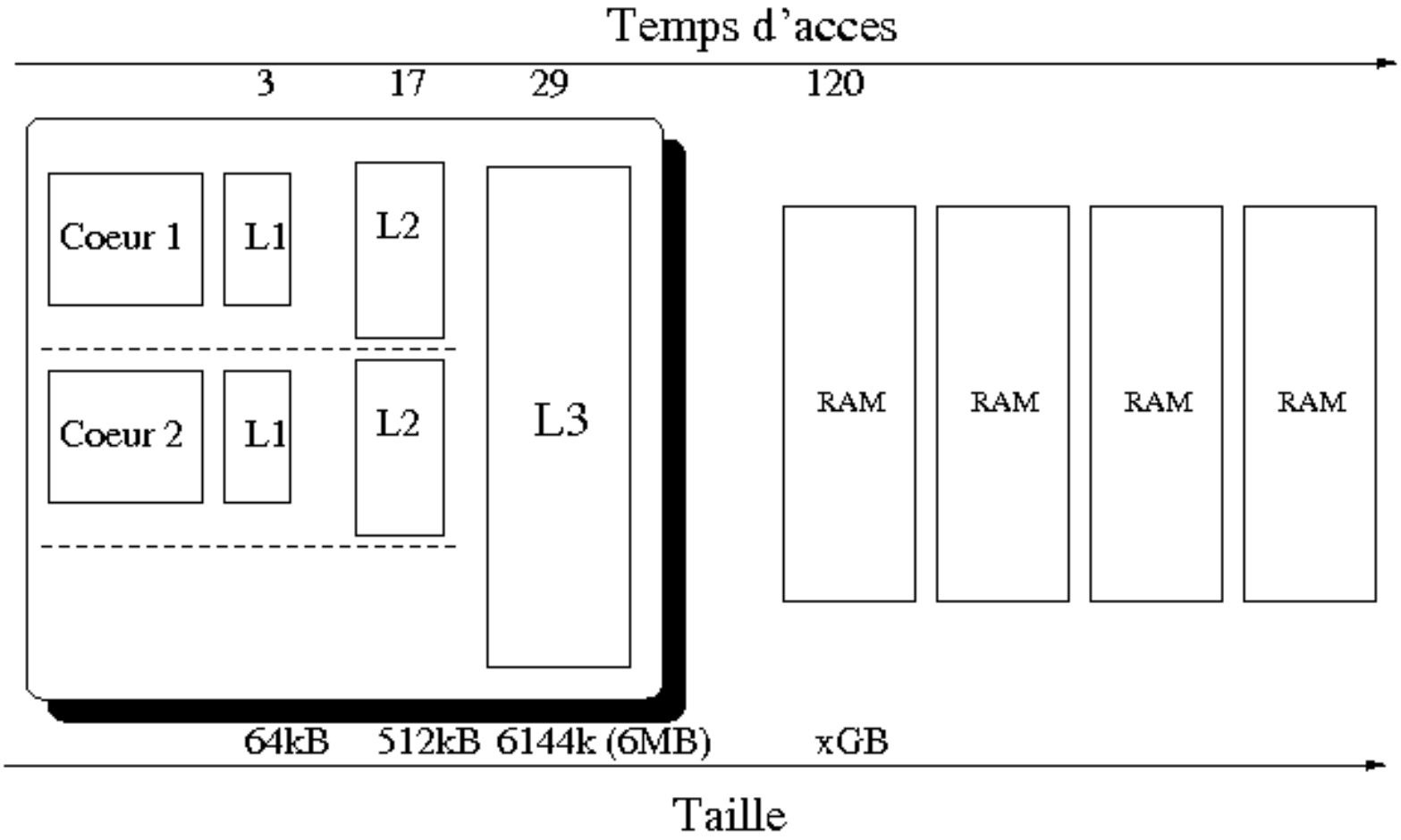
- Ai-je vraiment besoin de tous ces calculs ? $\Rightarrow$ penser pré-calculs, ré-utilisation des résultats, calcul sur des zéros
- Est-ce que j'utilise le bon algorithme ? Quelle est sa complexité (nombre d'opérations) ? En existe-t'il d'autres moins coûteux ?
- Que me coûte l'accès aux données ? $\Rightarrow$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

Mais je veux vraiment écrire du code ! II

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.

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



.

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

Augmentation du nombre de coeurs $\Rightarrow$ Limiter / utiliser au mieux l'accès mémoire

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.

$\Rightarrow$ 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.

Plan

Qu'a t'on appris ?

Nous savons maintenant :

- Déboguer un programme existant
- Mesurer les performances de nos codes ou des codes tiers
- Écrire des codes ne sollicitant pas trop la mémoire, gage de performance et scalabilité