Ecole Doctorale

–

Performances et Optimisations

Violaine Louvet [1]

[1] Institut Camille jordan - CNRS

2012-2013

Introduction

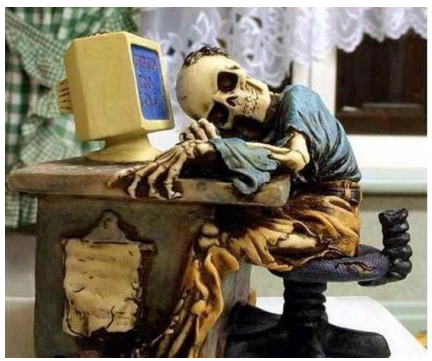
Optimisation ?

- Au sens lisibilité, portabilité, réutilisabilité
- Au sens améliorer les performances

Est-ce compatible ?

Optimiser pour :

- Exploiter au mieux l'architecture matérielle
- Réduire le temps de calcul
- Réduire l'empreinte mémoire
- Pouvoir faire tourner des calculs plus complexes, plus longs, plus gros

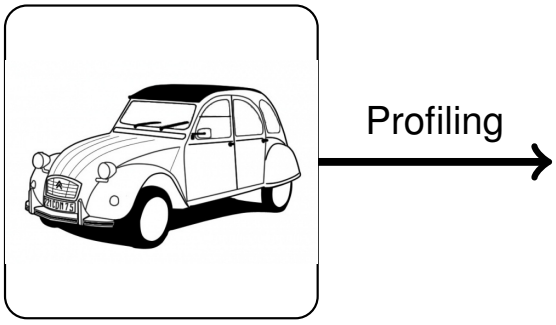


Stratégie

- Avoir un code qui fonctionne, et qui donne les résultats attendus

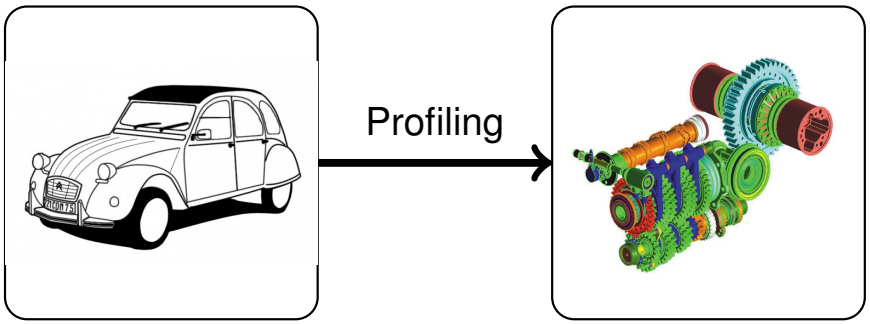
Stratégie

- Avoir un code qui fonctionne, et qui donne les résultats attendus
- Identifier les goulets d'étranglements, notamment en terme de temps de calcul
  - Les problèmes de mémoire entraînent la plupart du temps l'allongement du temps d'exécution



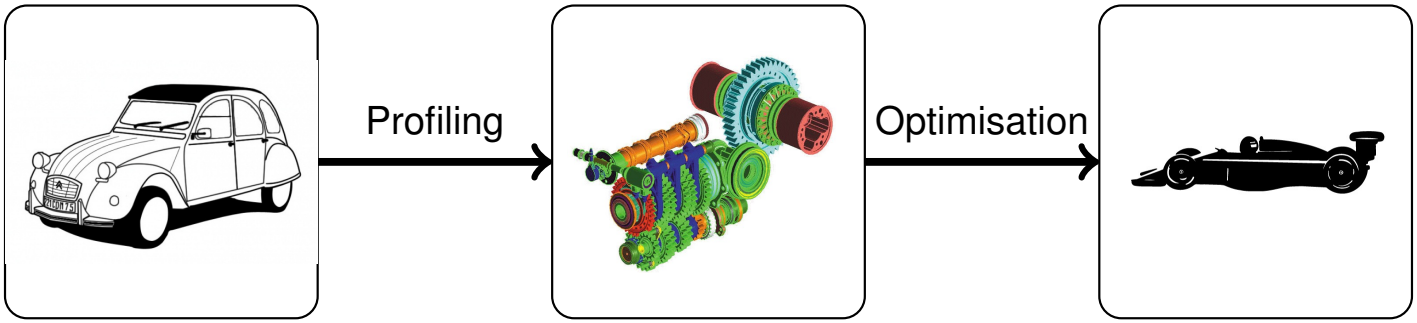
Stratégie

- Avoir un code qui fonctionne, et qui donne les résultats attendus
- Identifier les goulets d'étranglements, notamment en terme de temps de calcul
  - Les problèmes de mémoire entraînent la plupart du temps l'allongement du temps d'exécution
- Améliorer les parties les plus critiques



Stratégie

- Avoir un code qui fonctionne, et qui donne les résultats attendus
- Identifier les goulets d'étranglements, notamment en terme de temps de calcul
  - Les problèmes de mémoire entraînent la plupart du temps l'allongement du temps d'exécution
- Améliorer les parties les plus critiques
- Vérifier et valider le code au cours du processus d'optimisation



Objectifs de ce cours

Performances

- Déterminer les parties du code les plus coûteuses en temps.
- Déterminer les fonctions sur lesquelles faire porter l'effort d'optimisation.
- Savoir utiliser les outils de profiling.

Optimisations

- Réduire le temps de calcul.
- Réduire l'empreinte mémoire.
- Connaître les techniques d'optimisation permettant d'atteindre ces objectifs.

Sommaire

1 Profiling
- Définitions
- Mesure simple du temps
- Instrumentation statique
- Instrumentation dynamique
- Couverture du code

2 Optimisations
- Optimisations haut niveau
- Optimisations automatiques
- Optimisations manuelles : boucles et accès mémoire
- Allocation mémoire

Sommaire

1 Profiling
- Définitions
- Mesure simple du temps
- Instrumentation statique
- Instrumentation dynamique
- Couverture du code

2 Optimisations
- Optimisations haut niveau
- Optimisations automatiques
- Optimisations manuelles : boucles et accès mémoire
- Allocation mémoire

Sommaire

1 Profiling
- Définitions
- Mesure simple du temps
- Instrumentation statique
- Instrumentation dynamique
- Couverture du code

2 Optimisations
- Optimisations haut niveau
- Optimisations automatiques
- Optimisations manuelles : boucles et accès mémoire
- Allocation mémoire

Définitions

Outils de profiling

Outils indispensables pour optimiser de manière pertinente un code : ils permettent d'identifier les parties du code (hot spots, points chaud) sur lesquelles il va falloir travailler :

- Temps passé dans chaque partie du programme.
- Nombre d'appels des fonctions.
- Interaction du programme avec l'environnement : accès mémoire, accès concurrents aux données ...

Définitions

Différentes techniques

- Echantillonnage : l'exécution est échantillonnée régulièrement pour savoir quelles fonctions sont appelées. Pas d'intrusion mais résultat dépendant notamment de la fréquence d'échantillonnage. Plus la durée d'exécution est longue, plus les résultats sont précis.
- Instrumentation : le compilateur ajoute à chaque appel de fonction une fonction d'instrumentation qui va mesurer le temps d'appel, le nombre d'instructions exécutées ...
- Emulation : exécute le programme sur un processeur virtuel. Tout peut ainsi être mesuré de manière exacte mais cette méthode est très lente.

Sommaire

1 Profiling
- Définitions
- Mesure simple du temps
- Instrumentation statique
- Instrumentation dynamique
- Couverture du code

2 Optimisations
- Optimisations haut niveau
- Optimisations automatiques
- Optimisations manuelles : boucles et accès mémoire
- Allocation mémoire

Mesure simple du temps

Evaluer de façon non intrusive le temps d'exécution du programme : utilisation de la commande *time*.

```
$ time ./prog

real 0m2.671s
user 0m2.275s
sys 0m0.327s
```

- real : temps réel écoulé lors de l'exécution (temps passé, y compris lorsque le processus est en attente). Issu de l'appel système *gettimeofday*.
- user : temps d'utilisation CPU en mode utilisateur (les autres processus et les temps d'attente du processus ne sont pas comptabilisés). Issu des appels système *times* et *wait*.
- sys : temps d'utilisation CPU en mode noyau (appels système par opposition aux appels du code et des bilbiothèques utilisées). Issu des appels système *times* et *wait*.

✓ user+sys donne le temps CPU utilisé par le process, sur tous les CPU.

✓ Cas de plusieurs threads : user+sys $\geq$ real

Mesure simple du temps

- Evaluer plus finement de façon intrusive : utilisation de la routine système *gettimeofday* qui permet de connaître le temps passé dans une partie du programme.

```
#include <sys/time.h>

struct timeval start,end;
long int useconds;

gettimeofday (&start, (struct timezone*)0);
// Code a mesurer
gettimeofday (&end, (struct timezone*)0);
useconds = (end.tv_sec - start.tv_sec)*1000000
    + end.tv_usec - start.tv_usec;
```

```
$ time ./prog
Temps total : 2825498

real 0m2.671s
user 0m2.275s
sys 0m0.327s
```

Profiling

Techniques précédentes limitées à des évaluations ponctuelles sur le temps d'exécution
Nécessité d'automatiser et de pouvoir avoir d'autres données d'analyse

- Intrumentation statique : effectuée au plus tard à la compilation.
- Instrumentation dynamique : effectuée lors de l'exécution.

Données recueillies par le profiler

- Temps d'exécution de chaque fonction (profil plat).
- Graphe d'appels du programme.
- Temps inclusive : fonction appelante + fonction appelée.
- Temps exclusive : fonction appelante uniquement.

Sommaire

1 Profiling
- Définitions
- Mesure simple du temps
- Instrumentation statique
- Instrumentation dynamique
- Couverture du code

2 Optimisations
- Optimisations haut niveau
- Optimisations automatiques
- Optimisations manuelles : boucles et accès mémoire
- Allocation mémoire

gprof

gprof utilise à la fois des techniques d'échantillonnage et d'instrumentation
Instructions de profiling ajoutées à la compilation
Option de compilation : *-p -g*

- Chaque fonction est modifiée à la compilation pour mettre à jour les structures de données stockant la fonction appelante et le nombre d'appels.
- Le temps d'exécution est évalué par échantillonnage.
- Pour des fonctions non instrumentées (appel de bibliothèques par exemple), seule l'information du temps passé dans la fonction est disponible.

En pratique

- Compilation : *g++ -p -g prog.cpp -o prog*
- Exécution normale du programme : *./prog*
- Génération d'un fichier de données *gmon.out* dans le répertoire courant.
- Exécution de *gprof* : *gprof prog gmon.out*

Description des sorties de gprof : profil plat

% time : pourcentage du temps d'exécution total passé à exécuter cette fonction.

cumulative seconds : temps total cumulé que le processeur a passé à exécuter cette fonction, ajouté au temps passé à exécuter les fonctions précédentes dans le tableau.

self seconds : nombre de secondes passées à exécuter cette seule fonction.

calls : nombre d'appels de la fonction.

self ms/calls : nombre de millisecondes passées dans la fonction par appel.

total ms/calls : nombre de millisecondes passées dans cette fontion et ses enfants.

name : nom de la fonction

Description des sorties de gprof : graphe d'appels

- Différentes parties séparées par des tirets.
- Chaque partie débute par la ligne primaire :
  - Elle comprend en début de ligne un nombre entre crochets.
  - Elle se termine par le nom de la fonction concernée.
  - Les lignes précédentes décrivent les fonctions appelantes.
  - Les lignes suivantes décrivent les fonctions appelées : fonctions enfants.
  - Les entrées sont classée par temps passé dans la fonction et ses enfants.

index : Index référençant la fonction.

% time : pourcentage du temps passé dans la fonction, incluant les enfants.

self : temps total passé dans la fonction.

Children : temps total passé dans les appels aux enfants.

called : nombre d'appels de la fonction. Si appels récursif, la sortie est de la forme *n+m*, *n* désigne le nombre d'appels non récursifs et *m* le nombre d'appels récursifs.

name : nom de la fonction avec son index.

Profiling plus détaillé : Oprofile

Profiling ligne à ligne. Ne nécessite qu'une compilation avec option de debuggage.

En pratique

- Compilation : *g++ -g prog.cpp -o prog*
- Exécution du programme sous profiling : *operf ./prog*
- Premier niveau de profiling : *opreport –callgraph*
- Deuxième niveau de profiling : *opannotate –source*
- 1ère colonne : nbre d'échantillons de la ligne, 2ème colonne : % relatif par rapport à l'échantillonnage total.

```
  299   0.4644  :        for (int j = 0 ; j < m ; j++) {
  280   0.4349  : l = j + i * m;
 1596   2.4788  : arr2[i][j]=arr1[l]*log(i+1);
                :        }
                :
                :     for (int j = 0; j < m ; j++)
 1025   1.5920  :       for (int i = 0 ; i < n ; i++) {
  275   0.4271  : l = j + i * m;
20452  31.7647  : arr1[l]=arr2[i][j]*sin((2*i)/(j+1));
                :        }
```

Sommaire

1 Profiling
- Définitions
- Mesure simple du temps
- Instrumentation statique
- Instrumentation dynamique
- Couverture du code

2 Optimisations
- Optimisations haut niveau
- Optimisations automatiques
- Optimisations manuelles : boucles et accès mémoire
- Allocation mémoire

Valgrind

Suite d'outils de profilage et de débogage mémoire :

- Memcheck : un détecteur de fuites mémoires
- Cachegrind : un simulateur de caches
- Callgrind : un profileur

Fonctionnement

Le code est exécuté dans une machine virtuelle émulant un processeur équipé de nombreux outils d'analyse donnant des informations poussées sur le comportement d'un programme, qui ne pourraient être obtenues à l'aide d'un processeur matériel.

Quelques précisions

- Valgrind dégrade énormément les performances
- Ne surtout pas lancer une analyse Valgrind sur un programme complet long à exécuter

Débogage mémoire

En pratique

- Compilation du code :

```
$ g++ -g prog.cpp -o prog
```

- On peut vérifier que l'exécution du code se déroule correctement :

```
$ ./prog
Fin du programme
```

- Exécution sous valgrind :

```
$ valgrind --tool=memcheck ./prog
```

Débogage mémoire : évaluer les fuites mémoires

```
...
==4264==
==4264== HEAP SUMMARY:
==4264==     in use at exit: 0 bytes in 0 blocks
==4264==   total heap usage: 25 allocs, 25 frees, 24,020,009 bytes allocated
==4264==
==4264== All heap blocks were freed -- no leaks are possible
```

Fuites mémoire

- Si le programme est complexe, on peut lancer *vagrind* avec l'option *–leak-check=full* pour avoir plus d'informations sur la provenance des erreurs.
- A noter : le compilateur se charge de nettoyer correctement la mémoire, mais si ce n'est pas le cas, notre programme provoque des fuites mémoire

Utilisation du profiler

```
$ valgrind --tool=callgrind --dump-instr=yes ./prog
```

- *–dump-instr=yes* : permet d'enregistrer les instructions exécutées (facilite la comparaison avec le source).

- Génération d'un fichier *callgrind.out.numéro_pid*.
- Utilisation de l'outil KCacheGrind pour l'analyser.

Analyse de l'utilisation des caches

```
$ valgrind --tool=cachegrind ./prog
```

- Il est possible de spécifier la configuration des caches I1/D1/L2.

```
==29727==
==29727== I    refs:      3,358,368
==29727== I1   misses:        1,288
==29727== L2i  misses:        1,271
==29727== I1   miss rate:      0.03%
==29727== L2i  miss rate:      0.03%
==29727==
==29727== D    refs:      1,475,337  (1,149,931 rd   + 325,406 wr)
==29727== D1   misses:       16,516  (   12,225 rd   +   4,291 wr)
==29727== L2d  misses:        9,531  (    5,755 rd   +   3,776 wr)
==29727== D1   miss rate:       1.1% (       1.0%    +     1.3%  )
==29727== L2d  miss rate:       0.6% (       0.5%    +     1.1%  )
==29727==
==29727== L2 refs:           17,804  (   13,513 rd   +   4,291 wr)
==29727== L2 misses:         10,802  (    7,026 rd   +   3,776 wr)
==29727== L2 miss rate:         0.2% (       0.1%    +     1.1%  )
```

Sommaire

1 Profiling
- Définitions
- Mesure simple du temps
- Instrumentation statique
- Instrumentation dynamique
- Couverture du code

2 Optimisations
- Optimisations haut niveau
- Optimisations automatiques
- Optimisations manuelles : boucles et accès mémoire
- Allocation mémoire

gcov

- Outil de couverture de code.
- Permet de savoir quelles lignes de code sont effectivements exécutées et combien de fois elles le sont.
- Complémentaire de gprof.
- Permet notamment de vérifier que le jeu de test utilisé pour valider un programme est suffisant.

En pratique

- Compilation : *g++ -fprofile-arcs -ftest-coverage prog.cpp -o prog*
- Exécution normale du programme : *./prog*
- Génération de fichiers de données **.gcda* et **.gcno* dans le répertoire courant pour chaque fichier compilé avec les options précédentes.
- Exécution de *gcov* : *gcov prog.cpp*. Génération de fichiers de données **.gcov* : indexation des lignes par le nombre d'appels par ligne et leur numéro.

Description des sorties de gcov

- Réécriture du programme source en préfixant chaque ligne par le nombre de fois où elle a été exécutée. Les résultats sont clairement dépendants des données : des exécutions avec des données différentes donneront des résultats différents.
- Pour un branchement, pourcentage représentant le nombre de fois où la branche a été prise, divisé par le nombre de fois où le test a été exécuté. Si la branche n'a jamais été exécutée, message « never executed ».
- Pour un appel de fonction : nombre de sortie en fin de fonction sur le nombre d'appels (en général 100% sauf en cas d'exit dans la fonction).
- Les traces d'exécutions (contenues dans les fichiers .gcda) s'accumulent : ceci permet de mener des campagnes de statistiques sur un grand nombre de données, afin d'obtenir des informations plus fiables.

Sommaire

1 Profiling
- Définitions
- Mesure simple du temps
- Instrumentation statique
- Instrumentation dynamique
- Couverture du code

2 Optimisations
- Optimisations haut niveau
- Optimisations automatiques
- Optimisations manuelles : boucles et accès mémoire
- Allocation mémoire

Etat des lieux

Où en sommes nous ?

- Le programme tourne correctement en donnant des résultats raisonnables et attendus
- Nous avons une estimation du temps CPU du programme
- Nous avons identifié les « points chauds » au niveau du temps d'exécution, c'est-à-dire les endroits du programme consommateurs en temps CPU
- Nous avons évalué l'usage des caches lors d'une exécution

Etat des lieux

Où en sommes nous ?

- Le programme tourne correctement en donnant des résultats raisonnables et attendus
- Nous avons une estimation du temps CPU du programme
- Nous avons identifié les « points chauds » au niveau du temps d'exécution, c'est-à-dire les endroits du programme consommateurs en temps CPU
- Nous avons évalué l'usage des caches lors d'une exécution

Où va-t-on ?

- Améliorer le temps CPU
- Améliorer les accès mémoires

Etat des lieux

Où en sommes nous ?

- Le programme tourne correctement en donnant des résultats raisonnables et attendus
- Nous avons une estimation du temps CPU du programme
- Nous avons identifié les « points chauds » au niveau du temps d'exécution, c'est-à-dire les endroits du programme consommateurs en temps CPU
- Nous avons évalué l'usage des caches lors d'une exécution

Où va-t-on ?

- Améliorer le temps CPU
- Améliorer les accès mémoires

Attention

Règle 1 : Optimiser un code qui ne marche pas est inutile.
Règle 2 : Optimiser un code qui ne pose pas de problème de vitesse est inutile.
Axiome de base de ces règles : un code optimisé est beaucoup, beaucoup plus dur à maintenir. Et en général la première source de bogues.

Niveaux d'optimisation

Différents niveaux d'optimisations :

- niveau algorithmique et numérique : il est souvent plus efficace de choisir un algorithme, une structure de données ou une méthode numérique plus adaptés au problème que de s'attaquer à de l'optimisation bas niveau.
- niveau intermédiaire : choix du langage, utilisation de bibliothèques existantes, parallélisation.
- bas niveau : travail sur les sources en lien avec le profiler, instructions sse, options du compilateur, ...

Sommaire

1 Profiling
- Définitions
- Mesure simple du temps
- Instrumentation statique
- Instrumentation dynamique
- Couverture du code

2 Optimisations
- Optimisations haut niveau
- Optimisations automatiques
- Optimisations manuelles : boucles et accès mémoire
- Allocation mémoire

Optimisations haut niveau

- Méthode numérique : la méthode numérique choisit est-elle adaptée au problème, parallélisable, cohérente avec les structures de données ... ?
- Algorithme : évaluer la complexité de l'algorithme utilisé (nombre d'opérations), vérifier qu'il n'en existe pas de moins coûteux en temps, en mémoire, parallélisables ...
- Réutilisation : il y a de fortes chances que les opérations numériques nécessaires au code aient déjà été écrites : utiliser des bibliothèques existantes, souvent très optimisées et fiables.
- Accès aux données : évaluer la pertinence de la structure de données : l'accès est-il efficace, adapté à l'algorithme utilisé ?

Efficacité

La "règle" est que l'on peut peux gagner un facteur 1000 avec un bon algorithme et au maximum un facteur 10 avec du codage optimisé.

Sommaire

1 Profiling
- Définitions
- Mesure simple du temps
- Instrumentation statique
- Instrumentation dynamique
- Couverture du code

2 Optimisations
- Optimisations haut niveau
- Optimisations automatiques
- Optimisations manuelles : boucles et accès mémoire
- Allocation mémoire

Optimisations par le compilateur

- Idéalement, le compilateur optimise automatiquement le code en cherchant à utiliser au mieux les ressources internes du processeur.
- En pratique, il n'a pas toujours assez d'informations (notamment la taille des données connue au run-time, problème de dépendances ...).
- L'utilisation d'un autre compilateur permet :
  - de détecter des erreurs/warnings différents
  - de pratiquer une optimisation différente et donc d'apporter des gains de performance potentiels

Optimisations principales

- allocation optimale des registres, optimisation des accès mémoires
- élimination des redondances
- optimisation des boucles, en ne conservant à l'intérieur que ce qui est modifié
- optimisation du pipeline, utilisation du parallélisme d'instructions

Options de compilation

- -O*n*. Plus *n* est grand :
  - Plus l'optimisation est sophistiquée
  - Plus le temps de compilation est important
  - Plus la taille du code peut devenir grande
- Options de base :

  -O0 : aucune optimisation

  -O1 : optimisations visant à accélérer le code, en particulier quand il ne contient pas beaucoup de boucles.

  -O2 : *O1* + déroulage de boucles (considère notamment un aliasing strict). Augmente sensiblement le temps de compilation.

  -O3 : *O2* + transformation des boucles, des accès mémoire.

Attention

Certaines optimisations « agressives » peuvent modifier la précision des résultats (ordre des opérations par exemple).

Inlining par le compilateur

- Insertion du code d'une fonction à la place de l'appel
- Permet d'éviter l'overhead lié à l'appel
- Permet au compilateur d'avoir une vue plus globale du code et potentiellement d'optimiser davantage (notamment sur l'utilisation des registres)
- Le mot-clé *inline* (C++) est une indication pour le compilateur

Aliasing

Cas de l'utilisation de pointeurs

```
for (i=0;i<n;i++) a[i] = a[i]+b[i];
```

Si *a* et *b* ne se chevauche pas, la boucle peut être vectorisée mais le compilateur peut ne pas le savoir.

- On peut aider le compilateur : options *-fno-alias* (intel) ou *-fstrict-aliasing* (GNU) indiquent que 2 arguments de type pointeur dans tout le code ne pointent pas sur le même emplacement de la mémoire.
- Le standard Fortran interdit l'aliasing
- Spécification pour une boucle :

```
void add(float* a, float* b,int n) {
#pragma ivdep
  for (i=0;i<n;i++) a[i] = a[i]+b[i];
}
```

- Spécification pour des tableaux (standard C99, *__restrict__* en C++) :

```
void add(float* restrict a, float* restrict b,int n) {
  for (i=0;i<n;i++) a[i] = a[i]+b[i];
}
```

Vectorisation

- Compilateur GNU : *-ftree-vectorize* (par défaut avec *-O3*)
- Compilateur Intel : *-vec* et *-simd* (parallélisme de données)

Cilk+

Extension des langages C et C++ pour le support du parallélisme de données et de tâches.

- *#pragma simd* avant une boucle indique que le compilateur peut utiliser les registres vectoriels (SSE / AVX)

`http://cilkplus.org/`

Vérifier le travail du compilateur

- Mixer les sorties assembleur et le code source de départ :

```
g++ -g -c -Wa,-alh,-L fichier.cpp
```

- Vérifier les vectorisations des boucles :
  - GNU : *-ftree-vectorizer-verbose=n* avec $n \geq 2$
  - Intel : *-vec-report5*
- Dans le code assembleur, utilisation de commande de type *mulps, addps, xxxps ...*
- Le compilateur Intel fournit des diagnostics sur les optimisations réalisées avec l'option *-opt-report*

Sommaire

1 Profiling
- Définitions
- Mesure simple du temps
- Instrumentation statique
- Instrumentation dynamique
- Couverture du code

2 Optimisations
- Optimisations haut niveau
- Optimisations automatiques
- Optimisations manuelles : boucles et accès mémoire
- Allocation mémoire

Pourquoi est-ce critique ?

- Hiérarchie mémoire des processeurs actuels.
- En général, les boucles concentrent en peu de ligne un maximum de temps de calcul.
- Le compilateur fait ce qu'il peut mais il ne peut pas tout.
  - En particulier, si il ne peut pas déterminer les dépendances d'une boucle, elle ne sera pas optimisée
- Comportement régulier et répétitif : relativement simple à analyser statiquement.

Objectifs

- Améliorer l'accès aux données et donc leur localité spatiale.
- Aller plus vite.
- Réduire la surcharge dûe au contrôle des boucles.
- Investiguer les possibilités de vectorisation ou de parallélisation.

Le compilateur ne peut pas tout

- L'optimisation du compilateur est inhibée par :
  - Boucles sans compteur
  - Appel de sous-programmes dans une boucle
  - Condition de sortie non standard
  - Aliasing (pointeurs)
  - Tests à l'intérieur des boucles

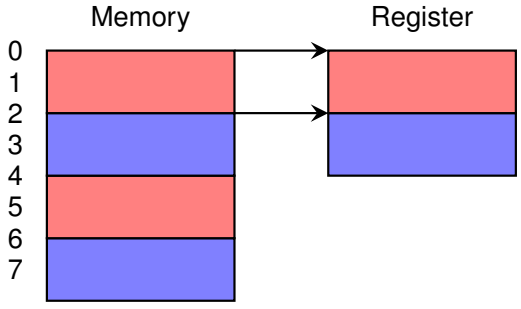
Aider le compilateur

- Il est donc important d'aider le compilateur
- En s'inspirant de ce qu'il cherche à faire :
  - Alignement de données
  - Transfomations de boucles
  - inlining
  - détection des invariants ...

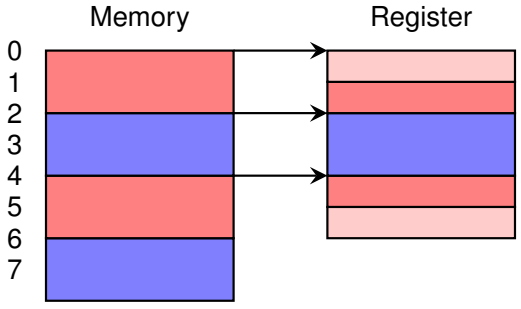
Alignement de données

- Positionner les instructions et données de manière efficace, quitte à perdre un peu d'espace mémoire
- L'accès à la mémoire se fait par morceaux de 2,4, 8, 16 ou 32 octets : granularité de l'accès mémoire
- Exemple : lecture de 4 octets avec une granularité mémoire de 2 octets

Lecture de 4 octets à l'adresse 0



Lecture de 4 octets à l'adresse 1



En pratique

- Quand on manipule des types de taille un multiple de 4 octets (32 bits) ou 8 octets (64 bits), ce qui est le cas la plupart du temps, pas trop de problèmes
- Le compilateur aligne correctement les données
- On peut forcer l'alignement par l'utilisation de directives type sse

Inlining et invariant des boucles

- Sortir les invariants des boucles :

Code original :

```
for (int i = 0 ; i < 10 ; i++)
   a[i] = b[i] + c/d;
```

Code optimal :

```
tmp = c/d;
for (int i = 0 ; i < 10 ; i++)
   a[i] = b[i] + tmp;
```

Coût d'un appel de fonction

Calcul des arguments, placement sur la pile, placement de l'@ de retour de la fonction, saut à la fonction, la fonction est exécutée, placement de la valeur de retour sur la pile, saut à l'@ de retour

- Inliner les appels de petites procédures dans les boucles : permet d'éviter l'overhead lié à l'appel de la routine.

Code original :

```
for (int i = 0 ; i < 10 ; i++)
   funct(a[i]);
```

Code optimal :

```
for (int i = 0 ; i < 10 ; i++)
   ... ! Corps de la routine funct
```

Déroulage de boucles

- Loop unrolling : réduit le coût de la gestion de la boucle (nombre de comparaison total, branchements, incréments) et augmente le parallélisme d'instructions potentiellement exploitable par les processeurs.

Code original :

```
for (int i = 0 ; i < 50 ; i++)
   a[i] = b[i];
```

Code optimal :

```
for (int i = 0 ; i < 50 ; i+=2) {
   a[i] = b[i];
   a[i+1] = b[i+1];
}
```

A noter

Le compilateur va probablement le faire tout seul en fonction des potentialités de l'architecture

Permutation de boucles

- Permutation des boucles imbriquées : alignement en mémoire des accès aux tableaux, réduction des défauts de cache.

Code original :

```
for (int j = 0 ; j < n ; j++)
   for (int i = 0 ; i < m ; i++)
      a[i][j] = 0.;
```

Code optimal :

```
for (int i = 0 ; i < m ; i++)
   for (int j = 0 ; j < n ; j++)
      a[i][j] = 0.;
```

- Attention, ce n'est pas toujours possible : dépendances des accès aux données dans les boucles

```
const int m=???,n=???,ioff=???,joff=???;

for (int j = 0 ; j < n ; j++)
   for (int i = 0 ; i < m ; i++)
      a[i][j] = funct(a[i+ioff][j+joff]);
```

Fortran

Le stockage en Fortran se fait ligne par ligne contrairement au C/C++ qui a un stockage colonne par colonne.

Fusion de boucles

- Fusion de boucles : améliore la localité temporelle des données et réduit le nombre de branchements.

Code original :

```
for (int i = 0 ; i < n ; i++)
   a[i] = ...;
for (int i = 0 ; i < n ; i++)
   ... = ... a[i] ...;
```

Code optimal : l'accès en lecture de *a[i]* se fait encore après son accès en écriture. La dépendance des données est respectée.

```
for (int i = 0 ; i < n ; i++)
   a[i] = ...;
   ... = ... a[i] ...;
```

Distribution de boucles

- Distribution de boucles : améliore la vectorisation et/ou la parallélisation des boucles et optimise l'usage du cache pour chaque boucle.

Code original :

```
for (int i = 1 ; i < n-1 ; i++) {
    a[i+1] = b[i-1] + c[i];
    b[i] = a[i]*k;
    c[i] = b[i] - 1;
}
```

Code optimal :

```
for (int i = 1 ; i < n-1 ; i++) {
    a[i+1] = b[i-1] + c[i];
    b[i] = a[i]*k;
}
for (int i = 1 ; i < n-1 ; i++) {
    c[i] = b[i] - 1;
}
```

Boucles avec condition

- Eliminer les conditions des boucles : pas de test ni de branchement dans le corps de la boucle.

Code original :

```
for (int i = 0 ; i < n ; i++) {
   a[i] = a[i] + b[i];
   if (expression) d[i] = 0.;
}
```

Code optimal :

```
if (expression) {
   for (int i = 0 ; i < n ; i++) {
      a[i] = a[i] + b[i];
      d[i] = 0.;
   }
} else {
   for (int i = 0 ; i < n ; i++)
      a[i] = a[i] + b[i];
}
```

Partage de l'espace d'itération

- Partage de la boucle : permet souvent d'éliminer des itérations problématiques.

Code original :

```
for (int i = 0 ; i < n ; i++) {
   a[i] = b[i] + c[i];
   if (i > 9) d[i] = a[i]+a[i-10];
}
```

Code optimal :

```
for (int i = 0 ; i < 10 ; i++)
   a[i] = b[i] + c[i];
for (int i = 10 ; i < n ; i++) {
   a[i] = b[i] + c[i];
   d[i] = a[i]+a[i-10];
}
```

Expansion des scalaires

- Expansion des scalaires : élimine de « fausses » dépendances.

Code original :

```
for (int i = 0 ; i < n ; i++) {
    t = a[i] + b[i];
    c[i] = t + 1;
}
```

Code optimal : la boucle peut maintenant être parallélisée.

```
double* t = new double[n];
for (int i = 0 ; i < n ; i++) {
    t[i] = a[i] + b[i];
    c[i] = t[i] + 1;
}
delete t;
```

Partitionnement de boucles

- Calcul par blocs : adapte la granularité (en gros la taille des blocs) à la hiérarchie mémoire.

Code original :

```
for (int i = 0 ; i < n ; i++)
  for (int j = 0 ; j < n ; j++)
    for (int k = 0 ; k < n ; k++)
      c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

Code optimal : on choisit S (stride) pour qu'un bloc de a, de b et de c tiennent dans le cache.

```
for (int ii = 0 ; ii < n ; ii+=S)
  for (int jj = 0 ; jj < n ; jj+=S)
    for (int kk = 0 ; kk < n ; kk+=S)
      for (int i = ii ; i < min(n,ii+S) ; i++)
        for (int j = jj ; j < min(n,jj+S) ; j++)
          for (int k = kk ; k < min(n,kk+S) ; k++)
            c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

Divers

- Eviter les conversions de type inutiles :

Code original :

```
double a[n];
for (int i=0; i<n; i++)
  a[i] = 2*a[i];
```

Code optimal :

```
double a[n];
for (int i=0; i<n; i++)
  a[i] = 2.0*a[i];
```

- Eliminer les sous-expressions :

Code original : calcul du polynôme $y = a + bx + cx^2 + dx^3$, 7 multiplications et 3 additions

```
y = a + b*x + c*x**2 + d*x**3;
```

Code optimal, 3 multiplications et 3 additions

```
y = a + (b + (c + d*x)*x)*x;
```

Sommaire

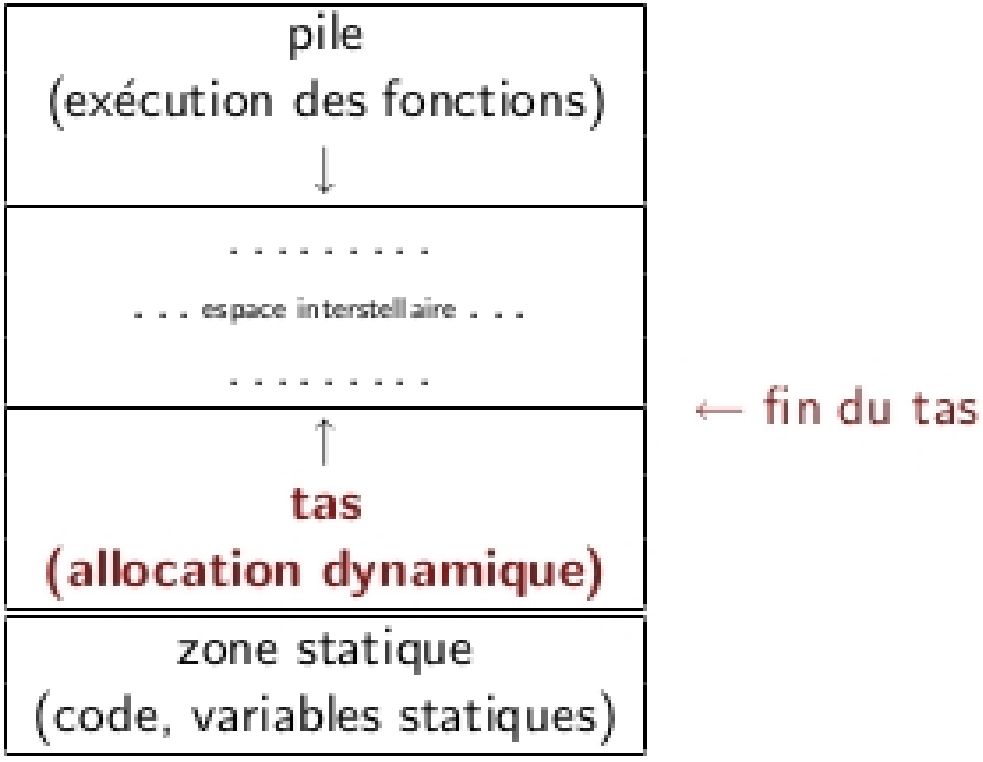
1 Profiling
- Définitions
- Mesure simple du temps
- Instrumentation statique
- Instrumentation dynamique
- Couverture du code

2 Optimisations
- Optimisations haut niveau
- Optimisations automatiques
- Optimisations manuelles : boucles et accès mémoire
- Allocation mémoire

3 types d'allocation mémoire

- Variables statiques (variables globales ou locales statiques) : Emplacement mémoire bien défini lors de la compilation.
- Variables automatiques (variables locales) :
  - Taille non définie a priori
  - Créées et détruites au fur et à mesure de l'exécution.
  - Alloc. sur la pile ("stack"), qui croît ou décroît selon la durée de vie des fonctions.
  - Gestion transparente pour le programmeur.
  - Mémoire correspondant typiquement à celle allouée par les appels de fonctions ou sous-programmes et à leurs variables locales non persistantes.
  - Gestion de la pile de type LIFO. Zone ordonnée.
- Variables dynamiques :
  - Taille non définie a priori.
  - Créées et détruites par demande explicite.
  - Alloc. sur le tas ("heap"), qui croît ou décroît selon les instructions du programmeur.
  - Mémoire correspondant typiquement aux allocations dynamique de mémoire (appels allocate en Fortran, calloc/malloc en C, new en C++).
  - Zone désordonnée.

Allocation mémoire

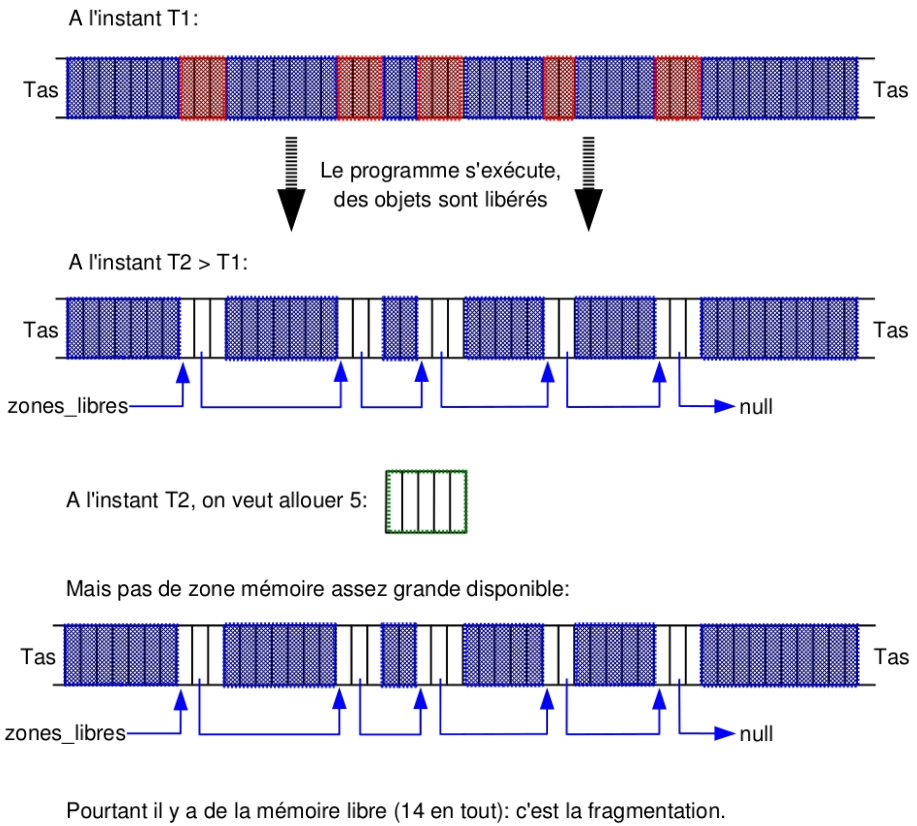


L'allocation sur la pile est beaucoup plus rapide que sur le tas : mouvement du pointeur de la pile (stack pointer) uniquement.

Allocation mémoire

Fragmentation du tas

Lors d'allocations et désallocations dynamiques répétées.



- Penser à libérer les espaces alloués non utilisés pour éviter les fuites mémoires
- Eviter les allocations répétées

Conclusion

D. Knuth

Premature optimization is the root of all evil (or at least most of it) in programming.

- Les optimisations de haut niveau sont les plus efficaces
- Il faut trouver un compromis entre optimisation et lisibilité/maintenabilité
- L'optimisation bas niveau rend délicate le portage des codes
- Il faut se focaliser sur ce qui est couteux en temps : cela ne sert à rien de chercher à optimiser une partie du code où on ne passe que peu de temps. Le gain final sera négligeable.
- Les Entrées/Sorties peuvent être des hot spots très importants dans certains codes.