Optimisation de codes

Anne Cadiou

Laboratoire de Mécanique des Fluides et d'Acoustique

Ateliers et séminaires pour l'informatique et le calcul scientifique
PMCS2I - LMFA
Vendredi 8 mars 2019

Introduction

Pourquoi optimiser son code ?

- Exploiter au mieux l'architecture matérielle
- Diminuer le temps de calcul (CPU time)
- Minimiser l'occupation mémoire
- Diminuer les communications inter-noeuds
- Réduire le temps passé à faire des I/O
- Exploiter au mieux les fonctionnalités d'un langages
- Rendre son code plus lisible, portable, évolutif

... des critères pas toujours compatibles.

Mise en oeuvre

Sur un code existant :

- S'assurer de la validation du code
- Profiler (temps d'exécution, utilisation de la mémoire, etc.)
- Modifier les parties les plus critiques
  - Vectorier (dérouler les boucles)
  - Revoir la structuration des données
  - Exploiter les caches
  - Utiliser des bibliothèques optimisées
  - Paralléliser
  - Revoir l'algorithme

Avant d'écrire le code :

- Choisir une méthode numérique adaptée au problème à traiter
- Définir une structure de données adaptée à la méthode numérique choisie
- Évaluer le type de parallélisation possible pour ces choix
- Choisir un algorithme adapté au problème à traiter
- Évaluer la complexité de l'algorithme (nombre d'opérations)
- Étudier les bibliothèques existantes à coupler au code

Étapes générales d'un code

- Initialisation
  - allocation mémoire,
  - lecture,
  - initialisation de variables, etc.
- Solveur
  - calcul d'opérations en virgules flottantes,
  - boucles, etc.
- Écriture et libération de la mémoire

Optimisation :

- Le solveur est généralement la partie la plus gourmande en calcul
- Les I/O et communications sont des parties où le processeur attend

Différents niveaux d'optimisation pour le calcul

- Automatique : à l'aide du compilateur
- Manuelle :
  - aider le compilateur,
  - exploiter le parallélisme,
  - s'appuyer sur les outils de profiling

Réduire le nombre total d'opérations dans le code, augmenter la localité spatiale et temporelle des données

Optimisation pour les I/O et communications

- Adapter la stratégie à l'architecture
- Masquer par du calcul
- Utiliser des bibliothèques standardisées et optimisées

Optimisation par le compilateur

Résolution d'un système linéaire par le méthode S.O.R. $64 \times 64 \times 64$ (2002)

Système	Cpu	Compilateur	Options	Time
Linux 2.4.7-10smp (Redhat 7.2)	Pentium IV Xeon 1.7Ghz Memory : 1024Mb RDRAM ECC PC860 Cache : 8K L1, 256K L2	Intel(R) Fortran Compiler for 32-bit applications Linux Version 5.0.1 Build	-O3 -tpp7 -axW -ipo	61s
Linux 2.4.7-10smp (Redhat 7.2)	Pentium IV Xeon 1.7Ghz Memory : 1024Mb RDRAM ECC PC860 Cache : 8K L1, 256K L2	Absoft f90	-O -B100	84s
Linux 2.4.7-10smp (Redhat 7.2)	Pentium IV Xeon 1.7Ghz Memory : 1024Mb RDRAM ECC PC860 Cache : 8K L1, 256K L2	GNU g77	-s -mcpu=pentiumpro -march=pentiumpro -mpentiumpro -O6 -frerun-cse-after-loop -fno-defer-pop -fschedule-insns -fomit-frame-pointer -fstrength-reduce -fforce-mem -fforce-addr -funroll-loops -freduce-all-givs -Wall	85s
Linux 2.4.7-10smp (Redhat 7.2)	Pentium IV Xeon 1.7Ghz Memory : 1024Mb RDRAM ECC PC860 Cache : 8K L1, 256K L2	Portland Group pgf90	-fast	88s
Alpha Server Compaq DS20 OSF 4.0F	Mémoire : 1.2 Gb Cache Prim. : ? Cache Second. : 4 Mb	DIGITAL f90	-O4 -tune ev6 -arch ev6 -fast	99s
Alpha Server Compaq 8200 OSF 4.0F	Mémoire : 2 Gb Cache Prim. : ? Cache Second. : 4 Mb	DIGITAL f90	-O4 -tune ev6 -arch ev6 -fast	150s
Linux 2.4.7-10smp (Redhat 7.2)	Pentium IV Xeon 1.7Ghz Memory : 1024Mb RDRAM ECC PC860 Cache : 8K L1, 256K L2	GNU g77	.	173s
Linux 2.4.2-2 (Redhat 7.2)	Pentium III Coppermine 993.33Mhz Memory : 512Mb Cache : 8K L1, 256K L2	Portland Group pgf90	-fast	366s
Linux 2.4.2-2 (Redhat 7.2)	Pentium III Coppermine 993.33Mhz Memory : 512Mb Cache : 8K L1, 256K L2	GNU g77	.	466s

(*d'après Violaine Louvet, LyonCalcul, 2012-2013*)

Le compilateur cherche à optimiser automatiquement le code. En pratique, il n'a pas toujours assez d'informations (notamment la taille des données connue au run-time, ...).

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

`-Olevel`

plus `level` est grand,

- plus l'optimisation est sophistiquée ;
- plus le temps de compilation est important ;
- plus la taille du code peut devenir grande.

`-O0` : aucune optimisation

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

`-O2` : O1 et déroulage de boucles ; augmente le temps de compilation

`-O3` : O2 et transformation des boucles, des accès mémoire, inlining

`-march=cpu_type` : génère les instructions adaptées au jeu d'instructions du processeur spécifié

Certaines optimisations peuvent modifier l'ordre des opérations et donc affecter les résultats (et leur reproductibilité).

Jeux d'instructions

```
program ProgALU

  implicit none
  integer :: n,ndim
  real :: x,y,deb,fin,i16

  ndim = 100000000
  x = 1.0
  y = 2.0
  deb=0.
  fin=0.

  call cpu_time(deb)
  n = 0
  do n = 1, ndim
    x = x + y**4
  end do
  call cpu_time(fin)

  print*, fin-deb

end program ProgALU
```

Résultat du calcul de

y^4	12.76 s
$y \times y \times y \times y$	0.51 s
$y/16$	0.57 s
$y \times 0.0625$	0.33 s

En utilisant les directives d'optimisation à la compilation **-O3** ce temps devient négligeable et equivalent.

Flops

La performance s'exprime en opérations à virgule flottante par seconde (Floating point Operations Per Second)
La puissance crête (point de vue théorique) mesure les performances des unités de calcul à virgule flottante (FPU) contenues dans le coeur

Exemple : Intel Core(TM) i7

4 coeurs, 2.6 GHz, registres vectoriels 16 (simple précision) ou 8 (double)

performance crête : $4 \times 2.6 \times 16 = 166.4$ GFlops

En pratique, la puissance d'une machine dépend

- des accès mémoire
- de la vitesse des bus
  (communication interne et réseau)
- du système d'exploitation
- de la charge de la machine
- la taille des mémoires caches

Impact de la complexité

```
program main

  implicit none

  integer :: n
  real*8, allocatable :: a(:),b(:),c(:)
  real*8, allocatable :: d(:)
  integer :: i,j

  ! initialisation
  n = 100000

  ! allocation
  allocate(a(n))
  allocate(b(n))
  allocate(c(n))
  allocate(d(n))

  call random_seed()
  call random_number(a)
  call random_number(b)
  call random_number(c)

  do i=1,n
    d(i) = 0.0D0
  end do
```

```
  ! calcul
  do j=1,n
    do i=1,n
      d(i) = d(i) + (a(i) + b(i)) * c(j)
    end do
  end do

  ! deallocation
  deallocate(a)
  deallocate(b)
  deallocate(c)
  deallocate(d)

end program main
```

Nombre d'opérations (flops) : $n \times n \times 3$

Complexité, flops et intensité

Complexité algorithmique :

nombre d'opérations

Estimation du temps d'exécution :

- dépend directement du nombre de cylces d'horloge effetuées par les opérations

  En supposant que les opérations élémentaires durent 2 cycles d'horloge, sur un processeur intel7 cadencé à 2.6 GHz (milliards de cycles d'horloge par seconde)
  $n \times n \times 3/(2.6 \times 1e9) \times 2.75 \sim 31$ secondes
  (ordre de grandeur)

```
real 0m36,514s
user 0m36,489s
sys 0m0,016s
```

- mais aussi du mode d'adressage du processeur (à la manière dont il va accéder à la mémoire), du nombre de caches, de la bande passante, etc.

Intensité arithmétique :

nombre d'opérations / quantité de mémoire échangée

```fortran
program main

  implicit none

  integer :: n
  real*8, allocatable :: a(:),b(:),c(:)
  real*8, allocatable :: s(:)
  real*8, allocatable :: d(:)
  integer :: i,j

  ! initialisation
  n = 100000

  ! allocation
  allocate(a(n))
  allocate(b(n))
  allocate(c(n))
  allocate(d(n))
  allocate(s(n))

  call random_seed()
  call random_number(a)
  call random_number(b)
  call random_number(c)
```

```
  ! preprocessing
  do i=1,n
    d(i) = 0.0D0
    s(i) = a(i) + b(i)
  end do

  ! calcul
  do j=1,n
    do i=1,n
      d(i) = d(i) + s(i) * c(j)
    end do
  end do

  ! deallocation
  deallocate(a)
  deallocate(b)
  deallocate(c)
  deallocate(d)
  deallocate(s)

end program main
```

Nombre d'opérations : $n + (2 \times n \times n)$

Réduction du temps de calcul global de $\sim 1/3$

```fortran
program main

  implicit none

  integer :: n
  real*8 :: sum_c
  real*8, allocatable :: a(:),b(:),c(:)
  real*8, allocatable :: s(:)
  real*8, allocatable :: d(:)
  integer :: i,j

  ! initialisation
  n = 100000

  ! allocation
  allocate(a(n))
  allocate(b(n))
  allocate(c(n))
  allocate(d(n))
  allocate(s(n))

  call random_seed()
  call random_number(a)
  call random_number(b)
  call random_number(c)
```

```
26
  sum_c = 0.0D0
28  do i=1,n
    d(i) = 0.0D0
30    s(i) = a(i) + b (i)
    sum_c = sum_c + c(i)
32  end do

34  ! calcul
  do i=1,n
36    d(i) = s(i) * sum_c
  end do
38
  ! deallocation
40  deallocate(a)
  deallocate(b)
42  deallocate(c)
  deallocate(d)
44  deallocate(s)

46 end program main
```

Nombre d'opérations : $3 \times n$
Temps de calcul : négligeable

```
real 0m0,033s
user 0m0,020s
sys 0m0,012s
```

Mémoires

Il existe plusieurs sortes de mémoires

- la RAM (Random Access Memory)
  - barrette physiques fixées à la carte mère
  - volatile (perd le contenu si on coupe le courant)
  - mémoire vive virtuelle : swap
- le disque dur
  - stockage des données
  - sauvegarde sur du long terme
- la mémoire cache
  - petite mémoire interne au processeur
  - d'accès très rapide

La bande passante caractérise un débit d'informations en octets par seconde
La latence désigne le temps minimum d'établissement de l'accès, mesuré en secondes

Hiérarchie

Type	Taille	Vitesse	Coût unitaire
registre	< 1 KB	< 1 ns	$$$$
SRAM On-chip	8 KB - 6 MB	< 10 ns	$$$
SRAM Off-chip	1 MB - 16 MB	< 20 ns	$$
DRAM	64 MB - 1 TB	< 100 ns	$
flash	64 MB - 32 GB	< 100 μs	c
disk	40 GB - 1 0B	< 20 ms	$\sim$

La bande passante et la vitesse augmentent avec la proximité avec le coeur
La latence augmente avec la taille
L'accès aux données est un des principaux facteurs limitant la performance
L'équilibre d'une machine se mesure par le ratio de la bande passante mémoire avec la performance crête

Fonctionnement des caches

- le cache est divisé en **lignes (ou blocs) de mots**
- 2 niveaux de granularité :
  - le CPU travaille sur des mots (par ex. de 32 ou 64 bits)
  - les transferts mémoire se font par ligne (ou blocs) (par ex. 256 octets)
- les lignes de caches sont organisées en ensembles à l'intérieur du cache, la taille de ces ensembles est constante et appelée le **degré d'associativité**
- exploitation de la **localité spatiale** : le cache contient des copies des mots par lignes de cache
- exploitation de la **localité temporelle** : choix judicieux des lignes de cache à retirer lorsqu'il faut ajouter une ligne à un cache déjà plein

Lorsque le processeur tente d'accéder à une information (instruction ou donnée), si l'information est dans le cache (**hit**), il n'y a pas d'attente, sinon (**miss**) le cache est chargé avec un bloc d'informations de la mémoire.

Principe de localité

Localité temporelle

Lorsqu'un programme accède à une donnée ou à une instruction, il est probable qu'il y accèdera à nouveau dans un futur proche

Localité spatiale

Lorsqu'un programme accède à une donnée ou à une instruction, il est probable qu'il accèdera ensuite aux données ou instructions voisines

```
subroutine sumVec(vec,n)

  integer :: n,vec(n)
  integer :: i,sum=0
  do n = 1, n
    sum = sum + vec(i)
  end do
end subroutine
```

- bonne localité spatiale des données du tableau par son accès séquentiel
- bonne localité temporelle de la données sum par son accès fréquent

Localité temporelle

```
do n = 1, 100000
  x = vec(n)
  y = vec(n+k)
end do
```

- conserver vec(n+k) en mémoire rapide jusqu'à sa prochaine utilisation à l'itération n+k
- si on veut exploiter les registres, il faut au moins k registres
- dans le cas général, on aura besoin de stocker d'autres données

Localité spatiale

manipulation de tableaux 2D : les données sont stockées dans un bloc mémoire contigu sous la forme d'un vecteur

```
ndim = 40000
do j = 1, ndim
  do i = 1, ndim
    y = a(i,j)*x(j)
  end do
end do
```

time : 3.68 secondes

```
ndim = 40000
do i = 1, ndim
  do j = 1, ndim
    y = a(i,j)*x(j)
  end do
end do
```

time : 31.61 secondes

- en **Fortran** stockage par colonnes (l'indice le plus à gauche varie le plus vite). donc si la matrice a une grande taille dans une direction, la placer en 1er indice et faire les boucles les plus internes sur le premier indice
- en **C** stockage par ligne. l'optimal est l'inverse du Fortran

respecter l'alignement en mémoire des tableaux, réduit les défauts de cache

Mémoire cache

Principe

Le processeur a besoin d'un débit soutenu en lecture d'instructions et de données pour ne pas attendre sans rien faire

Problème

La mémoire centrale qui stocke ces instructions et données est beaucoup trop lente pour assurer ce débit

Solution

Utiliser une mémoire très rapide intermédiaire entre la mémoire centrale et le processeur (cache) et exploiter sa localité

Organisation des caches

Il y a différents niveaux de cache.

- Le cache L_1 est le plus rapide, mais le plus petit. Il est scindé en deux parties, données et instructions, la notion de localité s'appiquant différemment pour les données et les instructions.
- Le cache L_{i+1} joue de rôle de cache pour le niveau L_i.

Certains niveaux de caches (souvent L_3 mais parfois L_2) peuvent être partagé entre plusieurs coeurs : c'est moins cher, mais potentiellement les accès sont plus lents car il peut y avoir des problèmes d'accès concurrents.

Aider le compilateur

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

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, etc.

Exemple avec le produit de deux matrices carrées

```
program test

  use mFunc

  implicit none

  integer :: n
  integer :: i,j,k
  real*8, allocatable, dimension(:,:) :: a,b,c
  real*8 :: deb_cpu_time,end_cpu_time

  ! data
  n = 1024

  ! allocate
  allocate(a(n,n))
  allocate(b(n,n))
  allocate(c(n,n))

  ! initialize
  call RANDOM_SEED()
  call cpu_time(deb_cpu_time)
  call RANDOM_NUMBER(a)
  call RANDOM_NUMBER(b)
  call init(n,c)
  call cpu_time(end_cpu_time)
  write(*,*) "Init :: elapsed CPU time :",end_cpu_time-deb_cpu_time
```

```
  ! product
  call cpu_time(deb_cpu_time)
  call prod_kji(n,a,b,c)
  call cpu_time(end_cpu_time)
  write(*,*) "kji :: elapsed CPU time :",end_cpu_time-deb_cpu_time
```

...

```
  ! deallocate
  deallocate(a)
  deallocate(b)
  deallocate(c)

end program test
```

$n = 1024$

$$C_{ij} = \sum_{k=1}^{k=n} A_{ik} \; B_{kj}$$

$n \times n \times n \times 2$ opérations

Algorithme proche de l'expression mathématique

```
subroutine prod_kji(n,a,b,c)

  implicit none

  ! declarations
  integer, intent(in) :: n
  real*8, dimension(:,:), intent(in) :: a,b
  real*8, dimension(:,:), intent(out) :: c

  ! local variables
  integer :: i,j,k

  do k=1,n
    do j=1,n
      do i=1,n
        c(i,j) = c(i,j) + a(i,k)*b(k,j)
      end do
    end do
  end do

end subroutine prod_kji
```

(respecte l'alignement pour le Fortran)

```
1.45s
```

Déroulage de boucles (loop unrolling)

```
subroutine prod_kji_unrolling2(n,a,b,c)

  implicit none
  integer, intent(in) :: n
  real*8, dimension(:,:), intent(in) :: a,b
  real*8, dimension(:,:), intent(out) :: c
  integer :: i,j,k
  real*8 :: s0,s1

  do j=1,n
    do k=1,n,2
      do i=1,n
        s0 = a(i,k)*b(k,j)
        s1 = a(i,k+1)*b(k+1,j)
        c(i,j) = c(i,j) + s0+s1
      end do
    end do
  end do

end subroutine prod_kji_unrolling2
```

Augmente le parallélisme d'instructions ; optimal dépendant de l'architecture (généralement réalisé par le compilateur)

```
0.88s
```

Découpage par blocs

```
subroutine prod_blocs(n,a,b,c)
  implicit none
  integer, intent(in) :: n
  real*8, dimension(:,:), intent(in) :: a,b
  real*8, dimension(:,:), intent(out) :: c
  integer :: i,j,k
  integer :: ii,jj,kk
  integer :: is,js,ks
  is = 8
  js = 8
  ks = 8
  do k=1,n,ks
    do j=1,n,js
      do i=1,n,is
        do kk=k,min(n,k+ks-1)
          do jj=j,min(n,j+js-1)
            do ii=i,min(n,i+is-1)
              c(ii,jj) = c(ii,jj) + a(ii,kk)*b(kk,jj)
            end do
          end do
        end do
      end do
    end do
  end do
end subroutine prod_blocs
```

```
1.84s
```

Transposée

```
subroutine prod_kjit(n,a,b,c)
  implicit none
  integer, intent(in) :: n
  real*8, dimension(:,:), intent(in) :: a,b
  real*8, dimension(:,:), intent(out) :: c

  integer :: i,j,k
  real*8, allocatable, dimension(:,:) :: bt

  allocate(bt(n,n))
  do k=1,n
    do j=1,n
      bt(j,k) = b(k,j)
    end do
  end do
  do k=1,n
    do j=1,n
      do i=1,n
        c(i,j) = c(i,j) + a(i,k)*bt(j,k)
      end do
    end do
  end do
  deallocate(bt)
end subroutine prod_kjit
```

```
0.88s
```

Uilisation de bibliothèques optimisées

```
subroutine prod_blas(n,a,b,c)
! product of matrice

  ! declarations
  implicit none
  integer, intent(in) :: n
  real*8, dimension(:,:), intent(in) :: a,b
  real*8, dimension(:,:), intent(out) :: c

  ! C = 1 x A x B + 0 x C
  call DGEMM('N','N',n,n,n,1.D0,a,n,b,n,0.D0,c,n)

end subroutine prod_blas
```

Bibliothèque optimisée BLAS (Basic Linear Algebra Subprogram)

```
0.17s
```

Distribution de boucles

Améliore la vectorisation et optimise l'usage du cache pour chaque boucle.

version initiale

```
do i = 2,n-1
  a(i+1) = b(i-1) + c(i)
  b(i) = a(i)*k
  c(i) = b(i)-1
end do
```

version optimisée

```
do i = 2,n-1
  a(i+1) = b(i-1) + c(i)
  b(i) = a(i)*k
end do
do i = 2,n-1
  c(i) = b(i)-1
end do
```

version initiale

```
do i = 1,n
  c(i) = 0.
  do j = 1,n
    c(i) = c(i) + a(i,j)*b(j)
  end do
end do
```

version vectorisable

```
do i = 1,n
  c(i) = 0.
end do
do j = 1,n
  do i = 1,n
    c(i) = c(i) + a(i,j)*b(j)
  end do
end do
```

Sortir les conditions des boucles

version initiale

```
do i = 1,n
  a(i) = b(i) + c(i)
  if (expression) d(i) = 0.
end do
```

version optimisée

```
if (expression) then
  do i = 1,n
    a(i) = b(i) + c(i)
    d(i) = 0.
  end do
else
  do i = 1,n
    a(i) = b(i) + c(i)
  end do
end if
```

version initiale

```
do i = 1,n
  a(i) = b(i) + c(i)
  if (i > 10) then
    d(i) = a(i) + a(i-10)
  end if
end do
```

version optimisée

```
do i = 1,10
  a(i) = b(i) + c(i)
end do
do i = 11,n
  a(i) = b(i) + c(i)
  d(i) = a(i) + a(i-10)
end do
```

Sortir les invariants des boucles

version initiale

```
do i = 1,n
  a(i) = b(i) + c/d
end do
```

version optimisée

```
cst = c/d
do i = 1,n
  a(i) = b(i) + cst
end do
```

version initiale

```
do k = 1,n
  c(k) = 2*(p-q)*(n-k+1)/(n*n+n)
end do
```

version optimisée

```
fact = 2*(p-q)
denom = n*n+n
do k = 1,n
  c(k) = fact*(n-k+1)/denom
end do
```

Inlining

Évite l'overhead lié à l'appel des petites procédures dans les boucles

version initiale

```
do i = 1,n
  call func(a(i))
end do
```

version optimisée

```
do i = 1,n
  ! corps de la routine func
  ! ...
end do
```

```
inline double Rand()
// random number between -1 and 1
{
    return 1-(random()*MAXRND);
}
```

```
for (int i=0; i<=NX; i++)
{
    U1k(i)=Complexe(Rand(),Rand());
    U2k(i)=Complexe(Rand(),Rand());
    U3k(i)=Complexe(Rand(),Rand());
}
```

Éviter les conversions de type inutiles

version initiale

```
real*8 :: a(n)
do i = 1,n
  a(i) = 2 * a(i)
end do
```

version optimisée

```
real*8 :: a(n)
do i = 1,n
  a(i) = 2.D0 * a(i)
end do
```

Compter les opérations dans les expressions

Pas uniquement un problème d'algorithme mais d'arithmétique
Éviter les opérations inutiles (ou trop coûteuses)

version initiale

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

7 multiplications et 3 additions
version initiale

```
up = val[(i-1)*n + j];
down = val[(i+1)*n + j];
left = val[i*n + j-1];
right= val[i*n + j+1];
sum = up + down + left + right;
```

3 multiplications

version optimisée

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

3 multiplications et 3 additions
version optimisée

```
int inj = i*n + j;
up = val[inj - n];
down = val[inj + n];
left = val[inj - 1];
right= val[inj + 1];
sum = up + down + left + right;
```

1 multiplication

Références

- Introduction à l'Optimisation de codes sur les Systèmes de Calcul Haute Performance, N. Renon, 2007
- Optimization techniques, S. Cozzini, 2008
- Code Optimization I: Machine Independent Optimizations, D.S. Fussell, 2011
- Performances et Optimisations, V. Louvet, 2011-2012
- Optimization techniques, A. Cassagne, 2014
- Optimisation séquentielle, IDRIS, 2016