

# Analyse de performances et optimisation de codes

---

Anne Cadiou

Atelier optimisation de codes

Groupe Calcul - DevLOG

Maison de la Simulation, vendredi 17 mai 2019

Laboratoire de Mécanique des Fluides et d'Acoustique



## 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 E/S
- Exploiter au mieux les fonctionnalités d'un langage
- Rendre son code plus lisible, portable, évolutif

... des critères pas toujours compatibles.

## 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
  - Vectoriser
  - Revoir la structuration des données
  - Exploiter les caches
  - Utiliser des bibliothèques optimisées
  - Paralléliser
  - Revoir l'algorithme

- 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 appeler dans le code

## Wikipédia

Analyser l'exécution d'une application, afin d'en connaître son comportement, d'évaluer les parties à optimiser

Sert à comprendre le comportement d'un code sur une architecture donnée, dans un environnement donné, afin d'exploiter au mieux toutes ses caractéristiques

## Questions associées

- Mon code utilise-t'il efficacement les ressources CPU ?
- La structure de mes données est-elle optimale ?
- Quelles sont les caractéristiques parallèles de mon programme ?
- Combien de mémoire effective utilise mon programme ?
- La gestion des E/S est-elle efficace ?

- 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

Généralement, les E/S et communications sont des parties où le processeur attend

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



## Optimisation par le compilateur

---

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.

`-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).

# 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 Me- mory : 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

```
program ProgALU
2
  implicit none
4  integer :: n,ndim
  real :: x,y,deb,fin
6
  ndim = 100000000
8  x = 1.0
  y = 2.0
10  deb=0.
  fin=0.
12
  call cpu_time(deb)
14  n = 0
  do n = 1, ndim
16    x = x + y**4
  end do
18  call cpu_time(fin)
20  print*, fin-deb
22 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

Calcul de

$$\sum_{j=1}^n \sum_{i=1}^n (a_i + b_i) c_j$$

Traduction littérale en Fortran

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

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



**Complexité algorithmique :**  
nombre d'opérations

Estimation du temps d'exécution :

- dépend directement du nombre de cycles d'horloge effectués par les opérations

En supposant que les opérations élémentaires durent 2.75 cycles d'horloge, sur un processeur intel7 cadencé à 2.6 GHz (milliards de cycles d'horloge par seconde)

$$n \times n \times 3 / (2.6 \times 1e9) \times 2.75 \sim 31 \text{ secondes}$$

(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

```
! 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
```

Nombre d'opérations :  $n + (2 \times n \times n)$   
Réduction du temps de calcul global  
de  $\sim 1/3$

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

! calcul
do i=1,n
  d(i) = s(i) * sum_c
end do
```

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

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

# Mémoire

---

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.

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 $\mu$ s	c
disk	40 GB - 1 0B	< 20 ms	~

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

- 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.

## 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)
2
   integer :: n,vec(n)
4   integer :: i,sum=0
   do n = 1, n
6     sum = sum + vec(i)
   end do
8 end subroutine
```

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

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

- conserver  $\text{vec}(n+k)$  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



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

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

time : 3.68 secondes

```
2 ndim = 40000  
do i = 1, ndim  
4   do j = 1, ndim  
   y = a(i,j)*x(j)  
   end do  
6 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

## 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

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'appliquant 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
- Transformations de boucles
- Inlining
- Détection des invariants, etc.

## Exemple avec le produit de deux matrices carrées

$n = 1024$

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

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

Algorithme proche de l'expression mathématique

(respecte l'alignement pour le Fortran)

```
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
```

1.45s

## Déroulage de boucles (loop unrolling)

```
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
```

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

0.88s

# Découpage par blocs

```
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
```

Adapter la taille des blocs pour qu'ils tiennent dans le cache

1.84s



```
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
```

0.88s

```
148 subroutine prod_blas(n,a,b,c)
    ! product of matrice
150
    ! declarations
152 implicit none
    integer, intent(in) :: n
154 real*8, dimension(:,:), intent(in) :: a,b
    real*8, dimension(:,:), intent(out) :: c
156
    ! C = 1 x A x B + 0 x C
158 call DGEMM('N','N',n,n,n,1.D0,a,n,b,n,0.D0,c,n)
160 end subroutine prod_blas
```

Bibliothèque optimisée BLAS (Basic Linear Algebra Subprogram)

0.17s

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

version initiale

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

version optimisée

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

version initiale

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

version vectorisable

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

# Sortir les conditions des boucles

## version initiale

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

## version optimisée

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

## version initiale

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

## version optimisée

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

# Sortir les invariants des boucles

version initiale

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

version initiale

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

version optimisée

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

version optimisée

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

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

version initiale

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

version optimisée

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

Inlining en C

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

```
for (int i=0; i<=NX; i++)
2 {
   U1k(i)=Complexe(Rand(),Rand());
4   U2k(i)=Complexe(Rand(),Rand());
   U3k(i)=Complexe(Rand(),Rand());
6 }
```

# Éviter les conversions de type inutiles

version initiale

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

version optimisée

```
real*8 :: a(n)
2 do i = 1,n
    a(i) = 2.D0 * a(i)
4 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];  
2 down = val[(i+1)*n + j];  
left = val[i*n + j-1];  
4 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;  
2 up = val[inj - n];  
down = val[inj + n];  
4 left = val[inj - 1];  
right = val[inj + 1];  
6 sum = up + down + left + right;
```

1 multiplication



## Profiling

---

- 1970s Outils d'analyse de performance basés sur une mesure d'intervalles de temps, développés sur IBM/360 et IBM/370 ;
- 1973 Outils d'analyse de codes sous Unix ;
  - prof : liste les instructions et le temps d'exécution ;
- 1974 Développements d'outils de simulation du jeu d'instructions (accès aux traces et autres mesures de performances) ;
- 1982 gprof : extension à l'analyse du graphe d'appel ;
- 1994 Développement d'ATOM (DEC) : bibliothèque d'instrumentation ajoutant des instructions à la compilation ;
- 2006 Essor de profilers de compilateurs JIT (Just In Time) pour les langages de haut niveau

## Principes

---

## Observer le comportement d'un programme à l'exécution

- Identifier les fonctions les plus consommatrices en CPU
- Connaître le temps passé à appeler les fonctions et bibliothèques
- Identifier la séquence des instructions
- Suivre l'allocation de la mémoire
- Identifier la matrice des communications, suivre l'équilibrage des charges
- Obtenir le coût des synchronisations, de l'ordonnanceur de tâches
- Mesurer les volume et débit des I/O

- Préparer le programme
    - recompiler avec une option spécifique
  - Exécuter le programme
    - éventuellement sous le contrôle d'un outil
    - enregistrer les statistiques (en mémoire ou dans un fichier)
  - Analyser les statistiques
- 
- Instrumentation statique (à la compilation)
  - Instrumentation dynamique (à l'exécution)

## Échantillonnage

- l'exécution du programme est suspendue périodiquement pour effectuer des mesures
- inférence statistique du comportement du programme
- fonctionne sans modification particulière du programme

## Instrumentation

- des capteurs sont insérés dans le code pour obtenir des informations détaillées
- biais important sur les petites fonctions
- nécessite la modification du code source

## Virtualisation

- exécution sur une machine virtuelle construite pour accéder à toutes les métriques
- méthode très lente

## Profiling

- obtention de temps d'exécution (statistiques, génération de résumés)
- souvent basé sur de l'échantillonnage

## Tracing

- obtention de cartographies, de chronologies (connexions, génération d'un journal de bord)
- souvent basé sur de l'instrumentation

- Profils plats
  - temps CPU passé dans chaque fonction
  - nombre de fois où une fonction est appelée
  - ⇒ utile pour connaître les fonctions les plus coûteuses
- Graphes d'appels
  - nombre de fois où une fonction est appelée par d'autres
  - nombre de fois où une fonction en appelle d'autres
  - ⇒ utile pour identifier la relation entre les fonctions
  - ⇒ suggère la suppression de certains appels
- Sources annotées
  - indique le nombre de fois où une ligne est exécutée



## Mesures

---

# Mesure non intrusive du temps

time donne de façon non intrusive le temps d'exécution d'une commande exécutée par le SHELL

```
acadiou@plume: ~$ sleep 5
```

```
acadiou@plume: ~$ time sleep 5 > cmd.out 2>&1
real 0m5.004s
user 0m0.004s
sys 0m0.000s
acadiou@plume: ~$ ls -l cmd.out
-rw-rw-r-- 1 acadiou acadiou 0 avril 17 17:44 cmd.out
```

real temps réel (walltime) écoulé pendant l'exécution

user temps CPU occupé par le programme

sys temps CPU utilisé par le programme pour faire des appels système

Le temps peut varier d'une exécution à l'autre (données statistiques)

`/usr/bin/time` commande unix qui retourne de façon non intrusive le temps d'exécution d'un programme

```
acadiou@plume: ~$ /usr/bin/time sleep 5 > prog.out 2>&1
acadiou@plume: ~$ cat prog.out

0.00user 0.00system 0:05.00elapsed 0%CPU (0avgtext+0avgdata 1832maxresident)k
0inputs+0outputs (0major+72minor)pagefaults 0swaps

acadiou@plume: ~$ /usr/bin/time -p sleep 5 > prog_format.out 2>&1
acadiou@plume: ~$ cat prog_format.out

real 5.00
user 0.00
sys 0.00
```

`man time`

perf commande unix qui retourne de façon non intrusive le temps d'exécution d'un programme

Il est nécessaire de modifier les fichiers de configuration par défaut pour permettre aux utilisateurs dans droits d'administration de les exploiter :

```
sudo sh -c 'echo -1 >/proc/sys/kernel/perf_event_paranoid'
```

### Liste des actions

```
perf list
```

### Statistiques

```
perf stat -e cycles,instructions,cache-misses
```

### Trace enregistrée dans un fichier perf.data

```
perf record -p PID
```

### Reporting (lecture de perf.data)

```
perf report  
perf report --stdio > perf.log
```

## Exemple d'usage

```
~$ perf stat -e cpu-clock sleep 5

Performance counter stats for 'sleep 5':

      2,042808 cpu-clock (msec) # 0,000 CPUs utilized

      5,003590297 seconds time elapsed
```

```
~$ perf stat sleep 5

Performance counter stats for 'sleep 5':

      0,800250 task-clock (msec) # 0,000 CPUs utilized
          1 context-switches # 0,001 M/sec
          0 cpu-migrations # 0,000 K/sec
          63 page-faults # 0,079 M/sec
1 187 237 cycles # 1,484 GHz
      909 456 instructions # 0,77 insn per cycle
      181 157 branches # 226,376 M/sec
         8 321 branch-misses # 4,59% of all branches

      5,001312184 seconds time elapsed
```

Appel de routines intrinsèques

Exemple en Fortran 90

- `dtime` Elapsed real time.
- `cpu_time` Elapsed CPU time. Fortran 90.

Temps CPU mesuré en secondes

```
real :: beg_cpu_time,end_cpu_time
call CPU_TIME(beg_cpu_time)
...
call CPU_TIME(end_cpu_time)
write(*,*) "Elapsed CPU time: ",end_cpu_time-beg_cpu_time
```

## Exemple en C

- `#include <time.h>` Processor time used by the process. Diviser par `CLOCKS_PER_SEC` pour une mesure en secondes.

```
#include <stdio.h>
#include <time.h> /* clock_t, clock, CLOCKS_PER_SEC */
clock_t clock_t beg_cpu_time,end_cpu_time;
double time_elapsed_s;
beg_cpu_time = clock();
...
end_cpu_time = clock();
time_elapsed_s = (end_cpu_time-beg_cpu_time)/(double) CLOCKS_PER_SEC;
printf("Elapsed CPU time (s): %f \n", time_elapsed_s);
```

## Exemple en Python

- `timeit.default_timer()` Python 3.3+
- `time.process_time()` Python 2.7+

```
import timeit

beg_cpu_time = timeit.default_timer()
...
end_cpu_time = timeit.default_timer()
print("Elapsed CPU time: ",end_cpu_time-beg_cpu_time)
```

Utiliser `%time` et `%timeit` avec `lpython` ou `Jupyter`

## Suivi interactif pendant l'exécution

- `top`
- `htop`
- `free -m`

## Outils de monitoring

- `gnome-system-monitor`
- `vmstat`
- `perf`



# Qu'est ce qui tourne ?

## top

- **h** affiche l'aide
- **m** affiche l'occupation mémoire (3 affichages)
- **M** trie par occupation mémoire
- **P** affiche les tâches sur les processus CPU
- **P** trie par processus CPU
- **k** **PID SIGNAL** tue le processus d'ID PID avec le signal SIGNAL (généralement 9)
- **q** quitter

alternative **htop** : suivre les instructions, en commençant par l'aide (F1)

```
free -m
```

```
total used free shared buffers cached
Mem: 258313 10840 149907 0 108 104596
-/+ buffers/cache: 3701 254611
Swap: 260511 127 260384
```

Afficher en gigabytes

```
free -g
```

et plus ...

```
free --tera
```

```
vmstat
```

```
procs -----memory----- --swap-- -----io---- -system-- -----cpu-----
 r b swpd free buff cache si so bi bo in cs us sy id wa st
 0 0 0 18803116 1819280 8749912 0 0 36 67 186 167 6 1 93 0 0
```

# Afficher la liste des processus

ps

```
PID TTY TIME CMD
6754 pts/28 00:00:00 bash
6767 pts/28 00:00:00 ps
```

## Options utiles

ps -edf |grep commande

ps aux

ps -A --forest

ps -u user

# Suivi pendant le calcul

Enregistrer l'occupation mémoire occupé par le processus 418 (thunderbird)

(6 enregistrements toutes les secondes)

```
pidstat -r -p 418 1 5 >> mem.log
```

```
Linux 4.4.0-119-generic (plume) 19/04/2018 _x86_64_ (4 CPU)

15:59:07 UID PID minflt/s majflt/s VSZ RSS %MEM Command
15:59:08 1000 418 0,00 0,00 2572484 486984 1,48 thunderbird
15:59:09 1000 418 201,00 0,00 2572484 487016 1,48 thunderbird
15:59:10 1000 418 0,00 0,00 2572484 487016 1,48 thunderbird
15:59:11 1000 418 0,00 0,00 2572484 487016 1,48 thunderbird
...
```

```
while sleep 1; do ps --pid 418 -o pcpu= -o pmem= -o rss=; done;
```

```
2.9 1.7 590352
2.9 1.7 590352
2.9 1.8 617256
2.9 1.8 617384
2.9 1.8 622252
2.9 1.8 621100
(...)
```

## Outils

---

## PAPI

<http://icl.cs.utk.edu/papi>

- interface multiplateforme de compteurs hardware (opérations et cycles, accès mémoire et caches)
- utilisée par TAU, SCALASCA, ompP, mpiP

## Score-P

<http://www.score-p.org>

- bibliothèque de profiling et trace pour les applications HPC
- utilisée par TAU, SCALASCA, Vampir, Periscope

## Extrac

<http://tools.bsc.es/extrac>

- bibliothèque d'instrumentation
- utilisée par Paraver

## Pour débiter

- gprof  
<https://sourceware.org/binutils/docs/gprof>
- Valgrind (Callgrind, Cachegrind, Massif)  
<http://valgrind.org/>

## et d'autres

- VTune (commercial)
- oprofile  
<http://oprofile.sourceforge.net/news/>
- ompP  
<http://www.ompp-tool.com>
- perfExpert  
<https://github.com/TACC/perfexpert>
- etc.

l'écosystème est très riche...

- Intel Advisor, Trace Analyzer and Collector (commercial)
- PerfSuite  
<http://perfsuite.ncsa.illinois.edu/>
- mpiP (Lightweight, Scalable MPI Profiling)  
<http://mpip.sourceforge.net>
  - résultat sous format texte
  - à ajouter lors de l'édition de liens
- IPM (Integrated Performance Monitoring)  
<http://ipm-hpc.sourceforge.net>
  - résultat sous format texte
  - sans recompilation
- Paraver  
<https://tools.bsc.es/paraver/>
  - basé sur Extrae



- HPCToolkit  
<http://www.hpctoolkit.org>
  - basé sur de l'échantillonnage statistique
- MAQAO (Modular Assembly Quality Analyzer and Optimizer)  
<http://www.maqao.org/>
  - dédié à l'analyse intra-noeud
  - sans recompilation
- TAU (Tuning and Analysis Utilities)  
<http://www.cs.uoregon.edu/research/tau>
  - basé sur PAPI
  - séquentiel, parallèle (MPI et multithread)
  - graphique et ligne de commande
- SCALASCA (SCalable performance Analysis of LARgre SCale Applications)  
<http://www.scalasca.org/>
  - basé sur PAPI, exploite Score-P
  - séquentiel, parallèle (MPI et multithread)
  - graphique et ligne de commande

- Vampir (commercial) <https://vampir.eu/>
- CUBE <http://www.scalasca.org/>
  - utilisé par SCALASCA, Score-P, PerfSuite, Marmot, ompP, etc.
- Paraver <https://tools.bsc.es/paraver/>
  - utilisé par Extrae
- PerfExplore PerfView <http://perfsuite.ncsa.illinois.edu/>
  - inclus dans PerfSuite
- etc.

## Darshan

<https://www.mcs.anl.gov/research/projects/darshan/>

- Sans recompilation, avec la variable LD\_PRELOAD
- Analyse les appels à Posix, MPI-IO, HDF5 et NetCDF
- Le code doit avoir un appel à la bibliothèque MPI (e.g. MPI\_Init et MPI\_Finalize)
- Compatible C, C++, Fortran

## Méthodologie d'analyse

---

## Préparation

- Préparer l'environnement
- Obtenir un cas de référence
- Instrumenter le code le cas échéant

## Mesure

- Collecter les données
- Aggréger les informations
- Se focaliser sur une partie (appel, mémoire, E/S)

## Analyse

- Calculer les métriques
- Identifier les problèmes affectant les performances

## Optimisation

- Modifier le code pour résoudre ou diminuer les problèmes

et reprendre la préparation et les mesures.

## Exemples

---

## Exemple basé sur gprof

<https://sourceware.org/binutils/docs-2.28/gprof/>

Somme de matrices

Comparaison de différentes implémentations contenues dans mFunc.f90

Compilation avec profiling

```
f95 -fdefault-real-8 -fdefault-double-8 -pg -c mFunc.f90
f95 -fdefault-real-8 -fdefault-double-8 -pg -c main.f90
f95 -fdefault-real-8 -fdefault-double-8 -pg mFunc.o main.o -o a.out
```

Exécution et génération du fichier de monitoring (gmon.out)

```
/usr/bin/time ./a.out > job.log

2.66user 0.09system 0:02.76elapsed 99%CPU (0avgtext+0avgdata 395084maxresident)k
0inputs+16outputs (0major+98409minor)pagefaults 0swaps
```

Analyse avec gprof

```
gprof ./a.out gmon.out > prof.log
```

prof.log contient le profil plat et le graphe d'appel sous forme de fichier texte



```
Flat profile:
2
Each sample counts as 0.01 seconds.
4  % cumulative self self total
   time seconds seconds calls s/call s/call name
6  67.46 1.26 1.26 1 1.26 1.26 __mfunc_MOD_summatijk
   27.84 1.78 0.52 1 0.52 0.52 __mfunc_MOD_summatjik
8   4.82 1.87 0.09 1 0.09 0.09 __mfunc_MOD_summatkji
   0.00 1.87 0.00 1 0.00 1.87 MAIN__
```

Le temps passé dans chaque subroutine est donné  
en pourcentage du temps CPU total et en secondes

Résultat équivalent en instrumentant le code avec l'appel à CPU\_TIME

```
ijk :: elapsed CPU time : 1.2160000000000002
jik :: elapsed CPU time : 0.5959999999999964
kji :: elapsed CPU time : 9.2000000000000526E-002
```

# légende

```
12 % the percentage of the total running time of the  
time program used by this function.  
  
14 cumulative a running sum of the number of seconds accounted  
seconds for by this function and those listed above it.  
  
16 self the number of seconds accounted for by this  
18 seconds function alone. This is the major sort for this  
listing.  
  
20 calls the number of times this function was invoked, if  
22 this function is profiled, else blank.  
  
24 self the average number of milliseconds spent in this  
ms/call function per call, if this function is profiled,  
26 else blank.  
  
28 total the average number of milliseconds spent in this  
ms/call function and its descendents per call, if this  
30 function is profiled, else blank.  
  
32 name the name of the function. This is the minor sort  
for this listing. The index shows the location of  
34 the function in the gprof listing. If the index is  
in parenthesis it shows where it would appear in  
36 the gprof listing if it were to be printed.
```

# Graphe d'appel

```
44      Call graph (explanation follows)
46
48      granularity: each sample hit covers 2 byte(s) for 0.53% of 1.87 seconds
50      index % time self children called name
51      0.00 1.87 1/1 main [2]
52      [1] 100.0 0.00 1.87 1 MAIN__ [1]
53          1.26 0.00 1/1 __mfunc_MOD_summatijk [3]
54          0.52 0.00 1/1 __mfunc_MOD_summatjik [4]
55          0.09 0.00 1/1 __mfunc_MOD_summatkji [5]
56      -----
57          <spontaneous>
58      [2] 100.0 0.00 1.87 main [2]
59          0.00 1.87 1/1 MAIN__ [1]
60      -----
61          1.26 0.00 1/1 MAIN__ [1]
62      [3] 67.4 1.26 0.00 1 __mfunc_MOD_summatijk [3]
63      -----
64          0.52 0.00 1/1 MAIN__ [1]
65      [4] 27.8 0.52 0.00 1 __mfunc_MOD_summatjik [4]
66      -----
67          0.09 0.00 1/1 MAIN__ [1]
68      [5] 4.8 0.09 0.00 1 __mfunc_MOD_summatkji [5]
69      -----
```

## (explanation follows)

70 This table describes the call tree of the program, and was sorted by  
the total amount of time spent in each function and its children.

72

74 Each entry in this table consists of several lines. The line with the  
index number at the left hand margin lists the current function.  
The lines above it list the functions that called this function,  
76 and the lines below it list the functions this one called.  
This line lists:

78     index A unique number given to each element of the table.  
Index numbers are sorted numerically.

80     The index number is printed next to every function name so  
it is easier to look up where the function is in the table.

82

84     % time This is the percentage of the 'total' time that was spent  
in this function and its children. Note that due to  
different viewpoints, functions excluded by options, etc,  
86 these numbers will NOT add up to 100%.

88     self This is the total amount of time spent in this function.

90     children This is the total amount of time propagated into this  
function by its children.

92

94     called This is the number of times the function was called.  
If the function called itself recursively, the number  
only includes non-recursive calls, and is followed by  
96 a '+' and the number of recursive calls.

98     name The name of the current function. The index number is  
printed after it. If the function is a member of a  
100 cycle, the cycle number is printed between the  
function's name and the index number.

102

104 For the function's parents, the fields have the following meanings:

106     self This is the amount of time that was propagated directly  
108     from the function into this parent.

110     children This is the amount of time that was propagated from  
112     the function's children into this parent.

114     called This is the number of times this parent called the  
116     function '/' the total number of times the function  
118     was called. Recursive calls to the function are not  
120     included in the number after the '/'.  
122

124     name This is the name of the parent. The parent's index  
126     number is printed after it. If the parent is a  
128     member of a cycle, the cycle number is printed between  
130     the name and the index number.

132 If the parents of the function cannot be determined, the word  
134 '<spontaneous>' is printed in the 'name' field, and all the other  
136 fields are blank.

# Options utiles

```
126 For the function's children, the fields have the following meanings:
128     self This is the amount of time that was propagated directly
129     from the child into the function.
130
131     children This is the amount of time that was propagated from the
132     child's children to the function.
133
134     called This is the number of times the function called
135     this child '/' the total number of times the child
136     was called. Recursive calls by the child are not
137     listed in the number after the '/'.
138
139     name This is the name of the child. The child's index
140     number is printed after it. If the child is a
141     member of a cycle, the cycle number is printed
142     between the name and the index number.
143
144 If there are any cycles (circles) in the call graph, there is an
145 entry for the cycle-as-a-whole. This entry shows who called the
146 cycle (as parents) and the members of the cycle (as children.)
147 The '+' recursive calls entry shows the number of function calls that
148 were internal to the cycle, and the calls entry for each member shows,
149 for that member, how many times it was called from other members of
150 the cycle.
```

- `-b` : pour ne pas sortir le texte explicatif
- `--static-call-graph`
- `--graph[=symspec]` : permet de définir la racine de l'arbre (`-q[symspec]`)
- `-A` : annote le code source

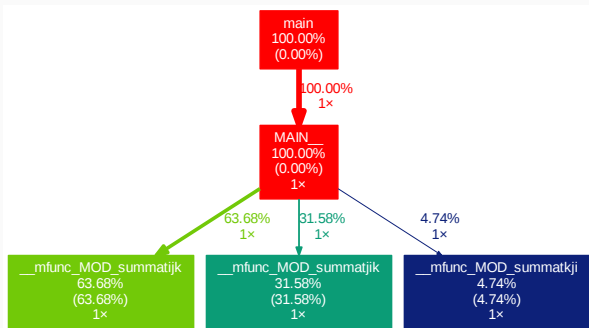
Basée sur graphviz et dot avec un petit utilitaire python gprof2dot.py

Générer le .dot

```
gprof ./a.out | ./gprof2dot.py > job.dot
```

Sortir l'image

```
dot -Tpdf job.dot -o output.pdf
```





gprof est un outil de base limité par certains aspects :

- Contenu des traces relativement sommaire
- Peu adapté au profiling d'applications parallèles, car trop sommaire : il peut juste générer un fichier `gmon.out` par processus

Pour le profiling parallèle et pour demander à chaque processus d'écrire un fichier de log, il suffit de définir une valeur à la variable d'environnement `GMON_OUT_PREFIX`.

Dans ce cas chaque processus PID produit un `gmon.out.PID`

Exemple :

```
export GMON_OUT_PREFIX='gmon.out'  
mpif90 -pg prof.f95 -o a.out  
mpirun -np 4 ./a.out
```

génère dans l'exemple les fichiers de trace

```
-rw-rw-r-- 1 acadiau acadiau 938 mai 4 10:01 gmon.out.9817  
-rw-rw-r-- 1 acadiau acadiau 938 mai 4 10:01 gmon.out.9818  
-rw-rw-r-- 1 acadiau acadiau 938 mai 4 10:01 gmon.out.9819  
-rw-rw-r-- 1 acadiau acadiau 938 mai 4 10:01 gmon.out.9820
```

à lire avec

```
gprof ./a.out gmon.out.*
```

# Exemple basé sur valgrind

<http://valgrind.org/docs>

```
valgrind --tool=callgrind --dump-instr=yes --simulate-cache=yes --collect-jumps=yes ./a.out >
  valgrind_prof.log 2>&1
```

Pour différer la création du rapport, ajouter l'option `--instr-atstart=no` et lancer dans un autre terminal afin d'activer le démarrage du profiling

```
callgrind_control -i on
```

```
-rw-rw-r-- 1 acadiou acadiou 1616 avril 19 12:14 valgrind_prof.log
-rw----- 1 acadiou acadiou 295162 avril 19 12:14 callgrind.out.14258
```

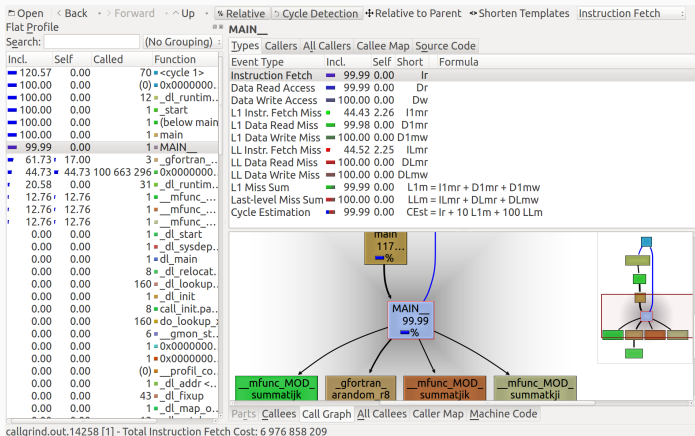
## Sortie du code

```
2 ijk :: elapsed CPU time : 1.2160000000000002
  jik :: elapsed CPU time : 0.59599999999999964
  kji :: elapsed CPU time : 9.20000000000000526E-002
```

# Fichier de profiling

```
2 ==14258== Callgrind, a call-graph generating cache profiler
3 ==14258== Copyright (C) 2002-2015, and GNU GPL'd, by Josef Weidendorfer et al.
4 ==14258== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
5 ==14258== Command: ./test.out
6 ==14258==
7 --14258-- warning: L3 cache found, using its data for the LL simulation.
8 ==14258== For interactive control, run 'callgrind_control -h'.
9 ==14258==
10 ==14258== Events : Ir Dr Dw I1mr D1mr D1mw I1Lmr D1Lmr D1Lmw
11 ==14258== Collected : 6976858209 1662625553 705269796 2386 71319689 41944841 2352 71306775 41944459
12 ==14258==
13 ==14258== I refs: 6,976,858,209
14 ==14258== I1 misses: 2,386
15 ==14258== L1i misses: 2,352
16 ==14258== I1 miss rate: 0.00%
17 ==14258== L1i miss rate: 0.00%
18 ==14258==
19 ==14258== D refs: 2,367,895,349 ( 1,662,625,553 rd + 705,269,796 wr)
20 ==14258== D1 misses: 113,264,530 ( 71,319,689 rd + 41,944,841 wr)
21 ==14258== L1d misses: 113,251,234 ( 71,306,775 rd + 41,944,459 wr)
22 ==14258== D1 miss rate: 4.8% ( 4.3% + 5.9% )
23 ==14258== L1d miss rate: 4.8% ( 4.3% + 5.9% )
24 ==14258==
25 ==14258== LL refs: 113,266,916 ( 71,322,075 rd + 41,944,841 wr)
26 ==14258== LL misses: 113,253,586 ( 71,309,127 rd + 41,944,459 wr)
27 ==14258== LL miss rate: 1.2% ( 0.8% + 5.9% )
```

# Sortie graphique avec Kcachegrind



valgrind dégrade énormément les performances

- Peu adapté à un programme long à exécuter
- Verbeux
- Vraiment très lent

## Exemple de profiling parallèle

---

NAS Parallel Benchmark (MPI/OpenMP)

<https://www.nas.nasa.gov/publications/npb.html>

## Cas BT-MZ

- Résolution des équations de Navier-Stokes compressibles 3D
  - Grille régulière
  - Avance de 200 pas de temps (solveur tri-diagonal par blocs)
- 
- NPB : The NAS Parallel Benchmarks
  - MZ : multi-zone
  - BT : Block Tri-diagonal solver

## Obtention du benchmark

```
~$ wget https://www.nas.nasa.gov/assets/npb/NPB3.4-MZ.tar.gz
~$ tar xvzf NPB3.4-MZ.tar.gz
~$ cd NPB3.4-MZ
NPB3.4-MZ$ cd NPB3.4-MZ-MPI
NPB3.4-MZ/NPB3.4-MZ-MPI$ ls
bin BT-MZ common config LU-MZ Makefile README README.install SP-MZ sys test_scripts
```

## Sélection des options pour le Makefile en fonction du calculateur

```
cp config/NAS.samples/make.def.gcc_mpich config/make.def
make bt-mz CLASS=B NPROCS=4
```



# Options de compilation

config/make.def

```
(...)  
#-----  
# This is the fortran compiler used for fortran programs  
#-----  
FC = mpif90  
# This links fortran programs; usually the same as ${FC}  
FLINK = $(FC)  
  
#-----  
# These macros are passed to the linker  
#-----  
F_LIB =  
  
#-----  
# These macros are passed to the compiler  
#-----  
F_INC =  
  
#-----  
# Global *compile time* flags for Fortran programs  
#-----  
FFLAGS = -O3 -fopenmp  
(...)
```

## Compilation du benchmark - sans profiling

```
NPB3.4-MZ/NPB3.4-MZ-MPI$ make bt-mz CLASS=B NPROCS=4
=====
= NAS PARALLEL BENCHMARKS 3.4 =
= MPI+OpenMP Multi-Zone Versions =
= MPI/Fortran =
=====

cd BT-MZ; make CLASS=B VERSION=
make[1]: Entering directory 'BT-MZ'
(...)
../sys/setparams bt-mz B
mpif90 -c -O3 -fopenmp bt_data.f
mpif90 -c -O3 -fopenmp mpinpb.f
mpif90 -c -O3 -fopenmp bt.f
(...)
mpif90 -O3 -fopenmp -o ../bin/bt-mz.B.x bt.o bt_data.o initialize.o exact_solution.o exact_rhs.o
set_constants.o adi.o rhs.o zone_setup.o x_solve.o y_solve.o exch_qbc.o solve_subs.o
z_solve.o add.o error.o verify.o setup_mpi.o mpinpb.o error_cond.o ../common/print_results
.o ../common/timers.o
make[1]: Leaving directory 'BT-MZ'
```

Exécutable dans bin/bt-mz.B.x

```
ls -lh bin
total 164K
-rwxr-xr-x 1 acadiau acadiau 162K avril 22 20:46 bt-mz.B.x
```

# Exécution du benchmark

## Environnement et exécution

```
export OMP_NUM_THREADS=2  
mpirun -np 2 ../bin/bt-mz.B.x
```

## Sortie

```
NAS Parallel Benchmarks (NPB3.4-MZ MPI+OpenMP) - BT-MZ Benchmark
```

```
Number of zones: 8 x 8  
Total mesh size: 304 x 208 x 17  
Iterations: 200 dt: 0.000300  
Number of active processes: 2
```

```
Use the default load factors  
Total number of threads: 4 ( 2.0 threads/process)
```

```
Calculated speedup = 4.00
```

```
Time step 1  
Time step 20  
(...)  
Time step 200  
Verification being performed for class B  
accuracy setting for epsilon = 0.100000000000000E-07  
(...)  
Verification Successful
```

```
BT-MZ Benchmark Completed.  
Class = B  
Size = 304x 208x 17  
Iterations = 200  
Time in seconds = 89.60  
Total processes = 2  
Total threads = 4  
Mop/s total = 6709.97  
Mop/s/thread = 1677.49  
Operation type = floating point
```

Compile options:

```
FC = mpif90  
FLINK = $(FC)  
F_LIB = (none)  
F_INC = (none)  
FFLAGS = -O3 -fopenmp  
FLINKFLAGS = $(FFLAGS)  
RAND = (none)  
(...)  
npb@nas.nasa.gov
```

## Recompilation

```
make clean
```

## Chargement de l'environnement

```
export PATH="/softs/scorep/bin:$PATH"  
export LD_LIBRARY_PATH="/softs/scorep/lib:$LD_LIBRARY_PATH"  
export PATH="/softs/scalasca-2.5/bin:$PATH"  
export LD_LIBRARY_PATH="/softs/scalasca-2.5/lib:$LD_LIBRARY_PATH"
```

## Modification du Makefile

```
(...)  
#-----  
# This is the fortran compiler used for fortran programs  
#-----  
FC = $(PREP) mpif90  
# This links fortran programs; usually the same as ${FC}  
FLINK = $(FC)
```

```
make bt-mz PREP=scorep CLASS=B NPROCS=4
```

```
ls -lh bin  
total 7,9M  
-rwxr-xr-x 1 acadiou acadiou 7,9M avril 22 20:47 bt-mz.B.x
```

# Ré-exécution du benchmark

```
export OMP_NUM_THREADS=2  
mpirun -np 2 ../bin/bt-mz.B.x
```

## Sortie

NAS Parallel Benchmarks (NPB3.4-MZ MPI+OpenMP) - BT-MZ Benchmark

Number of zones: 8 x 8  
Total mesh size: 304 x 208 x 17  
Iterations: 200 dt: 0.000300  
Number of active processes: 2

Use the default load factors  
Total number of threads: 4 ( 2.0 threads/process)

Calculated speedup = 4.00

Time step 1  
Time step 20  
(...)  
Time step 200  
Verification being performed for class B  
accuracy setting for epsilon = 0.10000000000000E-07  
(...)  
Verification Successful

## Coût supplémentaire en temps (overhead) lié à l'instrumentation :

```
BT-MZ Benchmark Completed.  
(...)  
Time in seconds = 100.58  
(...)  
npb@nas.nasa.gov
```

## Génération des fichiers de mesure

```
$ ls -l scorep-bt-mz/  
MANIFEST.md  
profile.cubex  
scorep.cfg
```

# Rapport sous forme de texte

```
$ scorep-score scorep-bt-mz/profile.cubex
```

```
Estimated aggregate size of event trace: 50MB
```

```
Estimated requirements for largest trace buffer (max_buf): 25MB
```

```
Estimated memory requirements (SCOREP_TOTAL_MEMORY): 29MB
```

```
(hint: When tracing set SCOREP_TOTAL_MEMORY=29MB to avoid intermediate flushes or reduce requirements using USR regions filters.)
```

```
flt type max_buf[B] visits time[s] time[%] time/visit[us] region
ALL 26,089,663 1,218,400 404.18 100.0 331.73 ALL
OMP 26,047,552 1,217,152 401.63 99.4 329.98 OMP
MPI 42,070 1,246 1.91 0.5 1529.89 MPI
```



# Détail par fonction

```
$ scorep-score -r scorep-bt-mz/profile.cubex

(...)
OMP 2,238,336 51,456 0.02 0.0 0.35 !$omp parallel @exch_qbc.f:217
OMP 2,238,336 51,456 0.02 0.0 0.33 !$omp parallel @exch_qbc.f:206
OMP 2,238,336 51,456 0.02 0.0 0.38 !$omp parallel @exch_qbc.f:256
OMP 2,238,336 51,456 0.02 0.0 0.35 !$omp parallel @exch_qbc.f:245
OMP 1,124,736 25,856 0.07 0.0 2.82 !$omp parallel @rhs.f:29
OMP 1,119,168 25,728 0.01 0.0 0.46 !$omp parallel @add.f:23
OMP 1,119,168 25,728 0.03 0.0 1.12 !$omp parallel @y_solve.f:44
OMP 1,119,168 25,728 0.03 0.0 1.13 !$omp parallel @x_solve.f:47
OMP 1,119,168 25,728 0.03 0.0 1.21 !$omp parallel @z_solve.f:44
OMP 668,928 51,456 0.04 0.0 0.87 !$omp implicit barrier @exch_qbc.f:215
(...)
```

```
(...)  
MPI 17,889 402 0.01 0.0 18.03 MPI_Isend  
MPI 17,889 402 0.00 0.0 7.97 MPI_Irecv  
OMP 11,136 256 0.00 0.0 2.95 !$omp parallel @initialize.f:23  
OMP 8,320 640 0.00 0.0 0.20 !$omp atomic @error.f:104  
OMP 8,320 640 0.00 0.0 0.18 !$omp atomic @error.f:51  
OMP 5,568 128 0.00 0.0 1.35 !$omp parallel @error.f:87  
OMP 5,568 128 0.00 0.0 2.10 !$omp parallel @exact_rhs.f:22  
OMP 5,568 128 0.00 0.0 1.34 !$omp parallel @error.f:28  
MPI 5,226 402 1.61 0.4 4013.78 MPI_Waitall  
(...)  
MPI 476 14 0.00 0.0 106.39 MPI_Bcast  
MPI 204 6 0.00 0.0 582.44 MPI_Reduce  
MPI 136 4 0.01 0.0 1256.30 MPI_Barrier  
MPI 68 2 0.00 0.0 86.88 MPI_Allgather  
MPI 52 4 0.00 0.0 0.75 MPI_Comm_rank  
MPI 52 4 0.00 0.0 3.03 MPI_Comm_size  
MPI 26 2 0.00 0.0 105.51 MPI_Comm_split
```

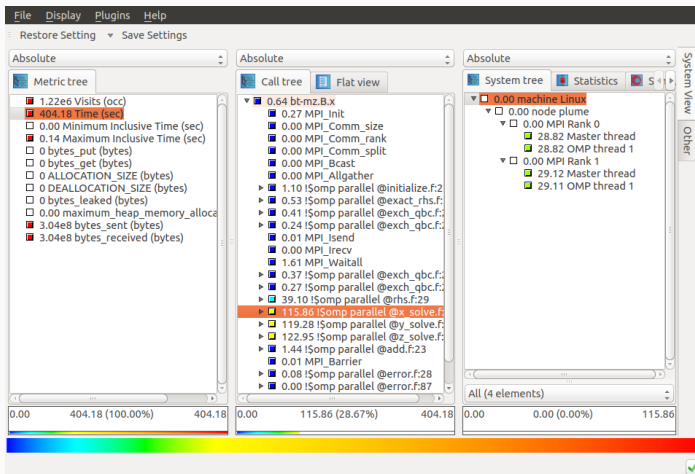
# Profil plat extrait avec CUBE

```
$ cube_stat -t 20 -p scorep-bt-mz/profile.cubex

cube::Region NumberOfCalls ExclusiveTime InclusiveTime
!$omp do @z_solve.f:51 25728 121.182131 121.182131
!$omp do @y_solve.f:51 25728 117.609398 117.609398
!$omp do @x_solve.f:53 25728 114.142487 114.142487
!$omp do @rhs.f:292 25856 10.884323 10.884323
!$omp do @rhs.f:182 25856 10.555631 10.555631
!$omp do @rhs.f:77 25856 10.358581 10.358581
!$omp do @rhs.f:34 25856 3.031725 3.031725
!$omp do @rhs.f:59 25856 2.408346 2.561018
!$omp implicit barrier @z_solve.f:427 25728 1.740977 1.740977
!$omp implicit barrier @x_solve.f:406 25728 1.689967 1.689967
!$omp implicit barrier @y_solve.f:405 25728 1.644672 1.644672
MPI_Waitall 402 1.613540 1.613540
!$omp do @add.f:23 25728 1.343585 1.425174
!$omp do @rhs.f:399 25856 1.040513 1.040513
!$omp do @initialize.f:51 256 0.995974 0.995974
bt-mz.B.x 2 0.639516 404.180371
!$omp implicit barrier @rhs.f:410 25856 0.573769 0.573769
!$omp do @exch_qbc.f:256 51456 0.352579 0.392707
!$omp do @exch_qbc.f:245 51456 0.306733 0.356091
MPI_Init 2 0.271824 0.271824
```

# Graphe d'appel avec CUBE

```
$ cube scorep-bt-mz/profile.cubex
```



```
scalasca -analyze mpirun -np 2 ../bin/bt-mz.B.x

S=C=A=N: Scalasca 2.5 runtime summarization
S=C=A=N: ./scorep_bt-mz_2x2_sum experiment archive
S=C=A=N: Mon Apr 22 22:11:51 2019: Collect start
(...)
Iterations = 200
Time in seconds = 100.02
(...)
S=C=A=N: Mon Apr 22 22:13:33 2019: Collect done (status=0) 102s
S=C=A=N: ./scorep_bt-mz_2x2_sum complete.
```

## Génération des fichiers de sortie

```
$ ls -l scorep_bt-mz_2x2_sum/
MANIFEST.md
profile.cubex
scorep.cfg
scorep.log
```

```
scalasca -examine -s scorep_bt-mz_2x2_sum
```

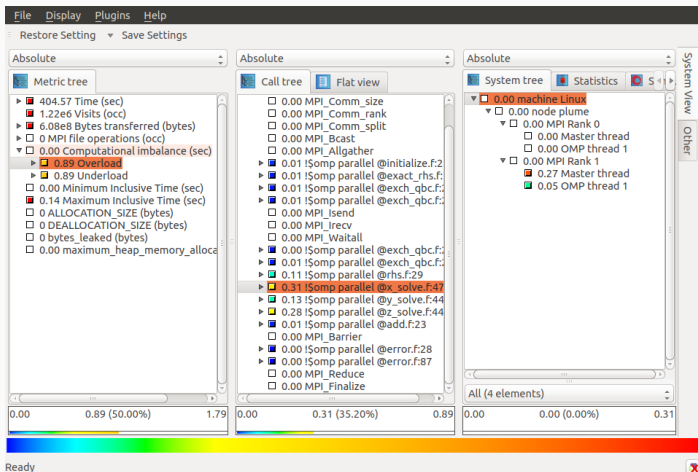
```
INFO: Post-processing runtime summarization report (profile.cubex)...
```

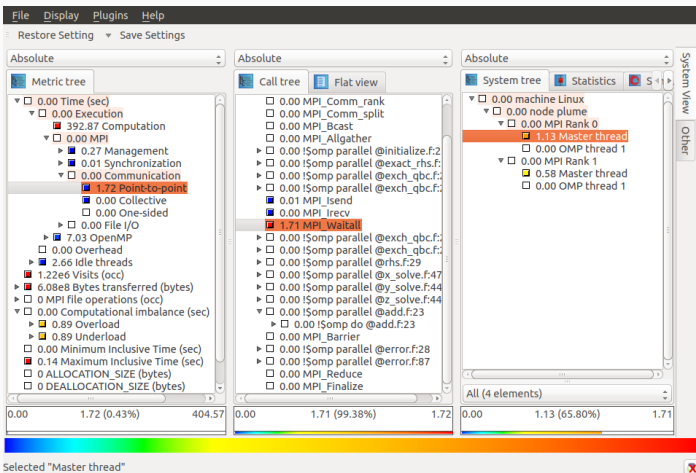
```
/softs/scorep/bin/scorep-score -r ./scorep_bt-mz_2x2_sum/profile.cubex > ./scorep_bt-mz_2x2_sum/  
scorep.score
```

```
INFO: Score report written to ./scorep_bt-mz_2x2_sum/scorep.score
```

# Visualisation avec CUBE

```
$ cube scorep-bt-mz/summary.cubex
```







## 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
- VI-HPS Tool guide
- NAS Benchmarks
- Tutoriels <https://computing.llnl.gov/>
- G. Markomanolis *Studying the behavior of parallel applications and identifying bottlenecks by using performance analysis tools*, École Méthodologie et outils d'optimisation en développement logiciel, IN2P3, 2012