Profiling

Principes, méthode et outils

Anne Cadiou

Laboratoire de Mécanique des Fluides et d'Acoustique

Séminaires et ateliers pour le calcul scientifique
PMCS2I - LMFA
Vendredi 4 mai 2018

Définition

Wikidépia

Analyser l'exécution d'un code afin d'en connaître son comportement

Objectifs

- Optimiser les temps de calcul
- Réduire la consommation mémoire ou I/O
- Estimer le comportement d'un code sur une architecture nouvelle (portage)
- Comprendre le comportement d'une architecture et son environnement

Historique

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 du profiling

Observer le comportement d'un programme à l'exécution

- Mesurer le temps d'exécution par morceaux de programme (fonction ou méthode)
- Compter les appels à une fonction ou méthode
- Suivre l'allocation de la mémoire
- Suivre l'usage du réseau le cas échéant

Différentes techniques

- Échantillonnage : l'exécution est échantillonnée régulièrement afin de connaître les fonctions appelées. Résultat dépendant de la fréquence d'échantillonnage.
- 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, etc.
- Émulation : l'exécution est faite sur une machine virtuelle construite pour accéder à toutes les métriques. Méthode très lente.

Différents niveaux

- Instrumenter le code
  - mise en place de mesures de temps
  - sortie sur les I/O, les communications
  - facile mais nécessite de modifier le code (et de charger les bibliothèques associées aux instruments)
- Profilage simple
  - identifier les fonctions les plus consommatrices en CPU
  - connaître le temps passé à appeler les fonctions et bibliothèques
  - identifier la séquence des instructions
- Profilage d'applications parallèles

  (*pas abordé dans cette session*)
  - obtenir la matrice des communications
  - coût des synchronisations, de l'ordonnanceur
  - améliorer l'équilibrage de charge
- Profilage des I/O
  - métrique du volume

Étapes de profiling de code

- 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)

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)

Commande UNIX

`/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 1832
    maxresident)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`

Exemple en Fortran 90

Appel de routines intrinsèques

- `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++

Appel de routines intrinsèques

- `#include <ctime>` Processor time used by the process. Diviser par `CLOCKS_PER_SEC` pour une mesure en seconde.
- `#include <chrono>` Elapsed CPU time. C++14.

Temps CPU mesuré en secondes

```
#include <iostream>
#include <ctime>        /* clock_t, clock, CLOCKS_PER_SEC */
std::clock_t beg_cpu_time,end_cpu_time;
float time_elapsed;
beg_cpu_time = std::clock();
...
end_cpu_time = std::clock();
time_elapsed = (end_cpu_time-beg_cpu_time)/CLOCKS_PER_SEC;
std::cout << "Elapsed CPU time: " << time_elapsed << " seconds\n" <<
      std::endl;
```

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 Ipython ou Jupyter

Suivi de l'occupation mémoire

Suivi interactif pendant l'exécution

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

Outils de monitoring

- `gnome-system-monitor`

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 commançant par l'aide (`F1`)

Mémoire utilisée

`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`

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
```

Enregistrer l'occupation mémoire occupé par le processus 814 (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
...
```

Quelques outils de base

Instrumentation du code et échantillonnage

- `gprof` gnu profiler, simple, pour débuter
- `oprofile`
- `perf-tools`
- `systemtap`

Quelques outils plus avancés

- `Intel Advisor`
- `TAU` (Tuning and Analysis Utilities)
  - basé sur PAPI
  - séquentiel, parallèle (MPI et Multithread)
  - graphique et ligne de commande
- `SCALASCA` (SCalable performance Analysis of LARgre SCale Applications)
  - basé sur PAPI
  - séquentiel, parallèle (MPI et Multithread)
  - graphique et ligne de commande
- `IPM` (Integrated Performance Monitoring)
  - résultat sous format texte
  - sans recompilation
- `mpiP` (Lightweight, Scalable MPI Profiling)
  - résultat sous format texte
  - à ajouter lors de l'édition de liens
- `ompP` (OpenMP Profiler)
  - basé sur PAPI

Exemple basé sur gprof

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

Somme de matrices

```
!> @file    test.f90
!> @author  anne cadiou
!> @brief   example for profiling

program test

  use mFunc

  implicit none

  integer :: ni,nj,nk
  integer :: i,j,k
  real, allocatable, dimension(:,:,:) :: x,y,z

  ! data
  ni = 256
  nj = 256
  nk = 256

  ! allocate
  allocate(x(ni,nj,nk))
  allocate(y(ni,nj,nk))
  allocate(z(ni,nj,nk))
```

```
24
   call RANDOM_SEED()
26
   ! initialize
28 call RANDOM_NUMBER(x)
   call RANDOM_NUMBER(y)
30 call RANDOM_NUMBER(z)

32 ! sum
   call sumMatijk(ni,nj,nk,x,y,z)
34
   ! sum reverse line and column
36 call sumMatjik(ni,nj,nk,x,y,z)

38 ! sum shorter access left index
   call sumMatkji(ni,nj,nk,x,y,z)
40
   ! deallocate
42 deallocate(x)
   deallocate(y)
44 deallocate(z)

46 end program test
```

```fortran
module mFunc
! Example of functions

  ! Declarations
  implicit none

contains

subroutine sumMatijk(ni,nj,nk,x,y,z)
! sum of matrices

  ! Declarations
  implicit none
  integer, intent(in) :: ni,nj,nk
  real, dimension(:,:,:), intent(in) :: x,y
  real, dimension(:,:,:), intent(out) :: z

  ! Local variables
  integer :: i,j,k

  ! Loop
  do i=1,ni
    do j=1,nj
      do k=1,nk
        z(i,j,k) = x(i,j,k)+y(i,j,k)
      end do
    end do
  end do

end subroutine sumMatijk
```

```fortran
subroutine sumMatjik(ni,nj,nk,x,y,z)
! sum of matrices

  ! Declarations
  implicit none
  integer, intent(in) :: ni,nj,nk
  real, dimension(:,:,:), intent(in) :: x,y
  real, dimension(:,:,:), intent(out) :: z

  ! Local variables
  integer :: i,j,k

  ! Loop
  do j=1,nj
    do i=1,ni
      do k=1,nk
        z(i,j,k) = x(i,j,k)+y(i,j,k)
      end do
    end do
  end do

end subroutine sumMatjik
```

```
subroutine sumMatkji(ni,nj,nk,x,y,z)
! sum of matrices

  ! Declarations
  implicit none
  integer, intent(in) :: ni,nj,nk
  real, dimension(:,:,:), intent(in) :: x,y
  real, dimension(:,:,:), intent(out) :: z

  ! Local variables
  integer :: i,j,k

  ! Loop
  do k=1,nk
    do j=1,nj
      do i=1,ni
        z(i,j,k) = x(i,j,k)+y(i,j,k)
      end do
    end do
  end do

end subroutine sumMatkji

end module mFunc
```

Compilation et exécution du code

Compilation avec profiling

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

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

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

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

Analyse avec `gprof`

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

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

Profil plat

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 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
  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

```
 %          the percentage of the total running time of the
time        program used by this function.

cumulative  a running sum of the number of seconds accounted
 seconds    for by this function and those listed above it.

 self       the number of seconds accounted for by this
seconds     function alone.  This is the major sort for this
            listing.

calls       the number of times this function was invoked, if
            this function is profiled, else blank.

 self       the average number of milliseconds spent in this
ms/call     function per call, if this function is profiled,
     else blank.

 total      the average number of milliseconds spent in this
ms/call     function and its descendents per call, if this
     function is profiled, else blank.

name        the name of the function.  This is the minor sort
            for this listing. The index shows the location of
     the function in the gprof listing. If the index is
     in parenthesis it shows where it would appear in
     the gprof listing if it were to be printed.
```

Graphe d'appel

```
44          Call graph (explanation follows)

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

(explanation follows)

```
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.

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,
and the lines below it list the functions this one called.
This line lists:
    index  A unique number given to each element of the table.
   Index numbers are sorted numerically.
   The index number is printed next to every function name so
   it is easier to look up where the function is in the table.

    % 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,
   these numbers will NOT add up to 100%.

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

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

    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
   a `+' and the number of recursive calls.

    name The name of the current function.  The index number is
   printed after it.  If the function is a member of a
   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
        from the function into this parent.
108
      children This is the amount of time that was propagated from
110     the function's children into this parent.

112      called This is the number of times this parent called the
        function '/' the total number of times the function
114     was called.  Recursive calls to the function are not
        included in the number after the '/'.
116
      name This is the name of the parent.  The parent's index
118     number is printed after it.  If the parent is a
        member of a cycle, the cycle number is printed between
120     the name and the index number.

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

Options utiles

```
For the function's children, the fields have the following meanings:

    self This is the amount of time that was propagated directly
   from the child into the function.

    children This is the amount of time that was propagated from the
   child's children to the function.

    called This is the number of times the function called
   this child '/' the total number of times the child
   was called.  Recursive calls by the child are not
   listed in the number after the '/'.

    name This is the name of the child.  The child's index
   number is printed after it.  If the child is a
   member of a cycle, the cycle number is printed
   between the name and the index number.

If there are any cycles (circles) in the call graph, there is an
entry for the cycle-as-a-whole.  This entry shows who called the
cycle (as parents) and the members of the cycle (as children.)
The '+' recursive calls entry shows the number of function calls that
were internal to the cycle, and the calls entry for each member shows,
for that member, how many times it was called from other members of
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` : annonte le code source

Version graphique

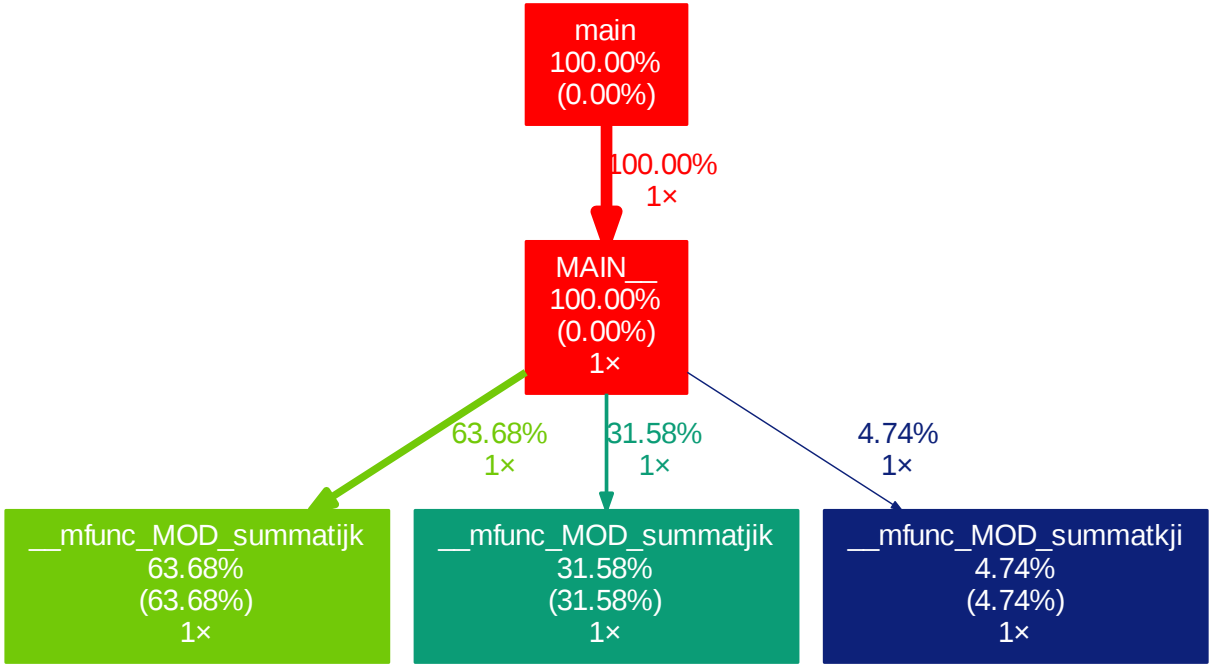
Basée sur `graphviz` et `dot` avec un petit utilitaire python `gprof2dot.py`

Générer le `.dot`

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

Sortir l'image

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


main
100.00%
(0.00%)
100.00%
1×
MAIN__
100.00%
(0.00%)
1×
63.68%
1×
31.58%
1×
4.74%
1×
__mfunc_MOD_summatijk
63.68%
(63.68%)
1×
__mfunc_MOD_summatjik
31.58%
(31.58%)
1×
__mfunc_MOD_summatkji
4.74%
(4.74%)
1×

Remarques

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 acadiou acadiou 938 mai    4 10:01 gmon.out.9817
-rw-rw-r-- 1 acadiou acadiou 938 mai    4 10:01 gmon.out.9818
-rw-rw-r-- 1 acadiou acadiou 938 mai    4 10:01 gmon.out.9819
-rw-rw-r-- 1 acadiou acadiou 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 ./test.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

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

Fichier de profiling

```
==14258== Callgrind, a call-graph generating cache profiler
==14258== Copyright (C) 2002-2015, and GNU GPL'd, by Josef Weidendorfer et al.
==14258== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==14258== Command: ./test.out
==14258==
--14258-- warning: L3 cache found, using its data for the LL simulation.
==14258== For interactive control, run 'callgrind_control -h'.
==14258==
==14258== Events    : Ir Dr Dw I1mr D1mr D1mw ILmr DLmr DLmw
==14258== Collected : 6976858209 1662625553 705269796 2386 71319689 41944841 2352 71306775
      41944459
==14258==
==14258== I   refs:      6,976,858,209
==14258== I1  misses:            2,386
==14258== LLi misses:            2,352
==14258== I1  miss rate:          0.00%
==14258== LLi miss rate:          0.00%
==14258==
==14258== D   refs:      2,367,895,349  (1,662,625,553 rd + 705,269,796 wr)
==14258== D1  misses:      113,264,530  (   71,319,689 rd +  41,944,841 wr)
==14258== LLd misses:      113,251,234  (   71,306,775 rd +  41,944,459 wr)
==14258== D1  miss rate:           4.8% (          4.3%   +         5.9%  )
==14258== LLd miss rate:           4.8% (          4.3%   +         5.9%  )
==14258==
==14258== LL refs:         113,266,916  (   71,322,075 rd +  41,944,841 wr)
==14258== LL misses:       113,253,586  (   71,309,127 rd +  41,944,459 wr)
==14258== LL miss rate:            1.2% (          0.8%   +         5.9%  )
```

Sortie graphique avec Kcachegrind

Open | Back | Forward | Up | % Relative | Cycle Detection | Relative to Parent | Shorten Templates | Instruction Fetch

Flat Profile

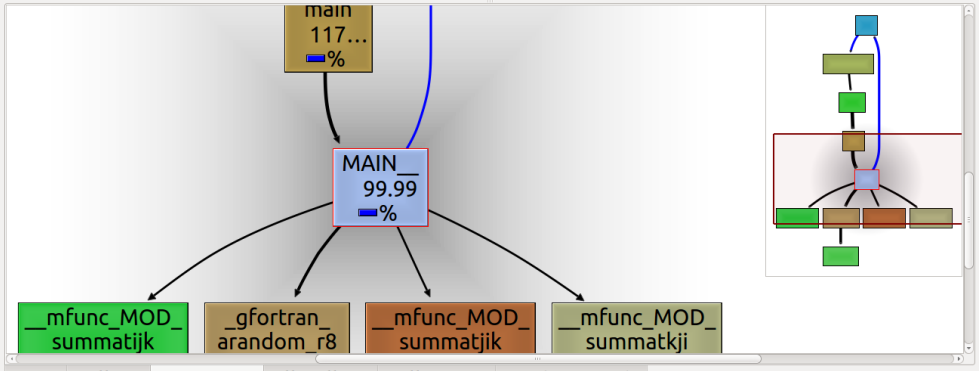
Search: (No Grouping)

Incl.	Self	Called	Function
120.57	0.00	70	<cycle 1>
100.00	0.00	(0)	0x0000000.
100.00	0.00	12	_dl_runtim..
100.00	0.00	1	_start
100.00	0.00	1	(below main
100.00	0.00	1	main
99.99	0.00	1	MAIN__
61.73	17.00	3	_gfortran_..
44.73	44.73	100 663 296	0x0000000.
20.58	0.00	31	_dl_runtim..
12.76	12.76	1	__mfunc_...
12.76	12.76	1	__mfunc_...
12.76	12.76	1	__mfunc_...
0.00	0.00	1	_dl_start
0.00	0.00	1	_dl_sysdep..
0.00	0.00	1	dl_main
0.00	0.00	8	_dl_relocat.
0.00	0.00	160	_dl_lookup..
0.00	0.00	1	_dl_init
0.00	0.00	8	call_init.pa..
0.00	0.00	160	do_lookup_
0.00	0.00	6	__gmon_st..
0.00	0.00	1	0x0000000.
0.00	0.00	1	0x0000000.
0.00	0.00	(0)	__profil_co..
0.00	0.00	1	_dl_addr <..
0.00	0.00	43	_dl_fixup
0.00	0.00	1	_dl_map_o..

MAIN__

Types | Callers | All Callers | Callee Map | Source Code

Event Type	Incl.	Self	Short	Formula
Instruction Fetch	99.99	0.00	Ir	
Data Read Access	99.99	0.00	Dr	
Data Write Access	100.00	0.00	Dw	
L1 Instr. Fetch Miss	44.43	2.26	I1mr	
L1 Data Read Miss	99.98	0.00	D1mr	
L1 Data Write Miss	100.00	0.00	D1mw	
LL Instr. Fetch Miss	44.52	2.25	ILmr	
LL Data Read Miss	100.00	0.00	DLmr	
LL Data Write Miss	100.00	0.00	DLmw	
L1 Miss Sum	99.99	0.00	L1m	= I1mr + D1mr + D1mw
Last-level Miss Sum	100.00	0.00	LLm	= ILmr + DLmr + DLmw
Cycle Estimation	99.99	0.00	CEst	= Ir + 10 L1m + 100 LLm



Parts | Callees | Call Graph | All Callees | Caller Map | Machine Code

callgrind.out.14258 [1] - Total Instruction Fetch Cost: 6 976 858 209

Remarques

`valgrind` dégrade énormément les performances

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

Référence des outils

- mpiP

  http://mpip.sourceforge.net
- ompP

  http://www.ompp-tool.com
- PAPI

  http://icl.cs.utk.edu/papi
- Tau

  http://www.cs.uoregon.edu/research/tau
- Scalasca

  http://www.fz-juelich.de/jsc/scalasca
- IMP

  http://ipm-hpc.sourceforge.net
- HPCToolkit

  http://www.hpctoolkit.org