

# Performance

## Analyse : Mesure et Optimisation

Romaric David

[david@unistra.fr](mailto:david@unistra.fr)

Direction Informatique

06/07/2014

changes  
spiritualidad  
insertion  
perspectives  
mutualisation  
reussite  
ouverture  
fondation  
CHEMISTRY  
spatium  
biology  
 $E = mc^2$   
RECHERCHE  
SYNERGIES  
COMPETENCES  
pi  
TECHNOLOGY  
doctorat  
cosmopolite  
ENSEIGNEMENT SUPÉRIEUR  
biotechnologies  
axiome  
mécanique  
management  
capitale  
droit  
excellence  
savoirs  
wissenschaft  
bibliothèques  
médecine  
tesis  
théologie  
gravitation  
idéaux  
connaissances  
musica  
langage  
INTERNATIONAL  
solution  
HEURISTIQUE  
partenariats  
HISTOIRE  
physique  
mécanique quantique  
insertion  
PLURIDISCIPLINARITÉ  
sciences  
gravitation  
humain  
mollécule  
ambition  
quantique  
MASTER  
cultures  
NETWORK

- ▶ Performance ? What is it all about ?
- ▶ Mesure
- ▶ Optimisation : CPU
- ▶ Optimisation : mémoire
- ▶ Conclusion

- ▶ En tant qu'utilisateur, nous sommes sensibles au **temps d'exécution** d'une application, à son **empreinte mémoire** et aux **ressources** qu'elle consomme. Ce cours présente quelques outils à notre disposition pour mesurer ces différentes grandeurs ⇒ **Profiling**.
- ▶ Le cours illustrera également quelques méthodes pour consommer moins de temps, mémoire et de ressources ⇒ **Optimisation**.
- ▶ Les programmes étudiés pour cet atelier sont des programmes séquentiels en C.

▶ Mesure du temps d'exécution

Outil externe : **time** ou **/usr/bin/time**

▶ **time** [executable]

- Temps total (wall-clock time)
- Temps en mode utilisateur (user)
- Temps passé dans les appels système (sys).  
Exemple : printf, read, write, ...
- Wall-clock = user + sys + temps d'attente i/o

▶ Appel système équivalent : **getrusage**

▶ Ceci ne donne qu'une vision globale du déroulé

- ▶ Connectez-vous sur vos PC
- ▶ `wget http://hpc-web.u-strasbg.fr/optim.tgz`
- ▶ Décompressez l'archive
- ▶ Vous obtenez un répertoire `optim_base`

# TP 1 : Temps d'exécution

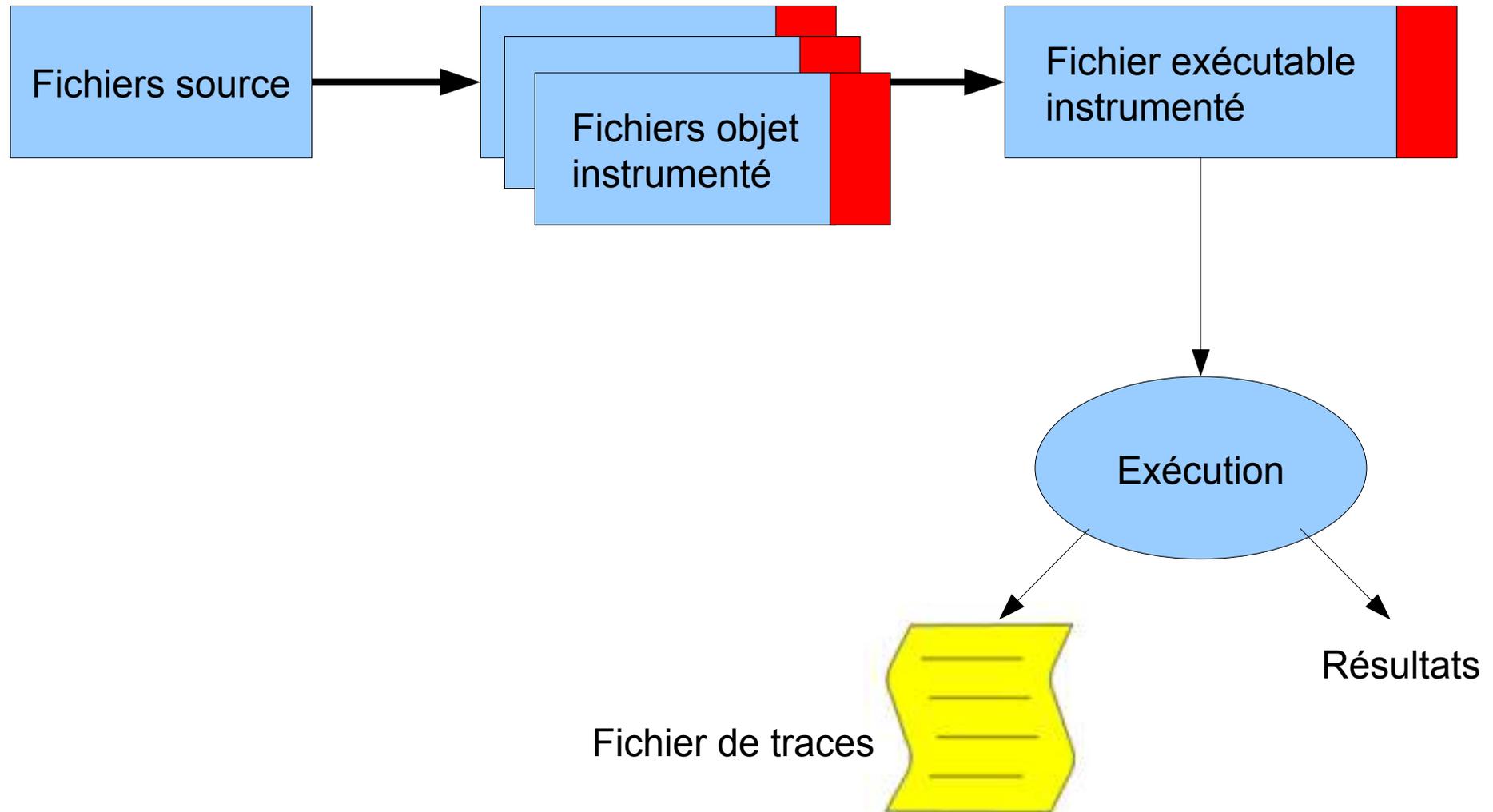
Unistra - 07/07/2014

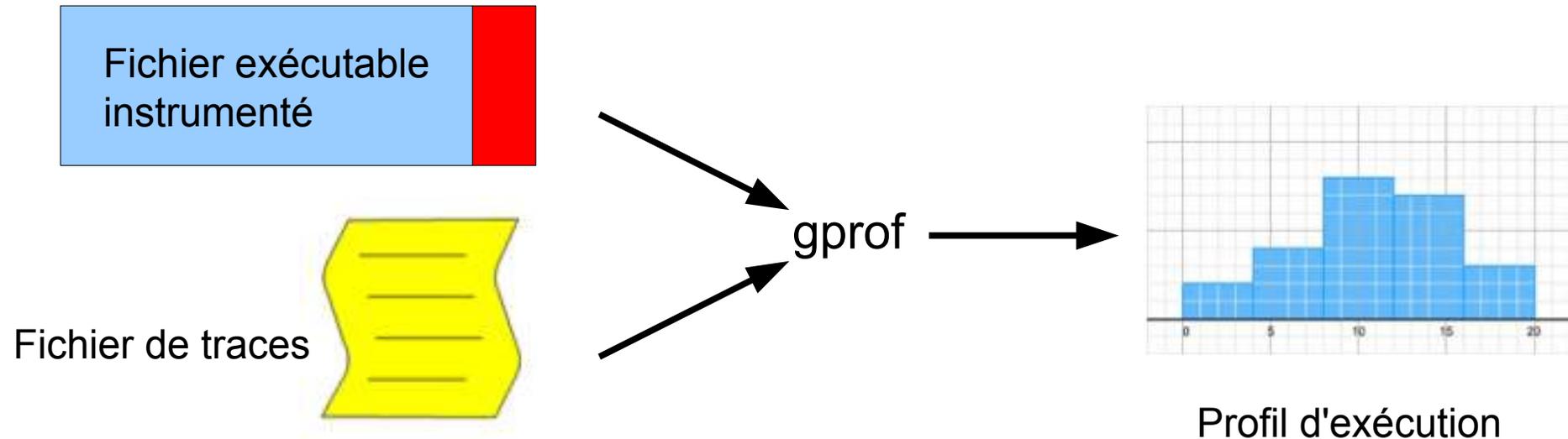
- ▶ Nous utilisons le fichier `optim_base/jv/jv.c`
- ▶ Compilez ce programme
- ▶ Exécutez le programme par `time ./executable`
- ▶ Que peut-on dire du temps passé en I/O ?



- ▶ À la compilation, il est possible d'instrumenter automatiquement le code, pour mesurer le temps passé routine par routine.
- ▶ Par échantillonnage à l'exécution, le programme examinera la fonction dans laquelle il se trouve et lui attribuera le quantum de temps écoulé
- ▶ Un profil est généré à la fin de l'exécution
- ▶ Ce profil est analysé avec gprof

# Mesure : 1 première loupe





► Dans le profil d'exécution, on retrouve :

- Le temps passé dans chaque fonction, en intégrant les fonctions appelées : **temps inclusif** (100% pour main)
- Le temps passé dans chaque fonction sans tenir compte des fonctions appelées : **temps exclusif**

- ▶ Nous utilisons le fichier `/optim_base/jv/jv.c`
- ▶ Compilez ce programme avec l'option d'instrumentation (`-pg`)
- ▶ Exécutez le programme, vous devriez obtenir un fichier `gmon.out`
- ▶ Analysez ce fichier avec `gprof` :  
`gprof ./a.out gmon.out`
- ▶ Examinons la sortie en détails (votre sortie peut varier)

## ► Premiers chiffres : profil plat

```
rdavid@jdevt7b:/usr/local/atelier/jv$ gprof ./a.out gmon.out  
Flat profile:
```

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
89.11	55.62	55.62	5	11.12	11.12	evolution
9.52	61.56	5.94	4	1.49	1.49	faffiche
1.38	62.42	0.86				main

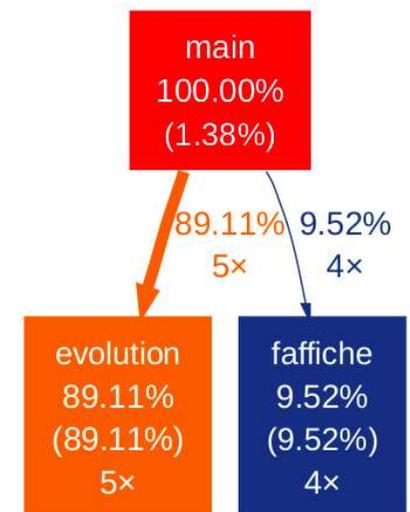
- Le profil est dit plat car chaque fonction est représentée au même niveau

## ▶ Deuxièmes chiffres : graphe d'appel

index	% time	self	children	called	name
					<spontaneous>
[1]	100.0	0.86	61.56		main [1]
		55.62	0.00	5/5	evolution [2]
		5.94	0.00	4/4	faffiche [3]
-----					
[2]	89.1	55.62	0.00	5	main [1]
		55.62	0.00	5	evolution [2]
-----					
[3]	9.5	5.94	0.00	4	main [1]
		5.94	0.00	4	faffiche [3]
-----					

- ▶ Ceci nous permet de retracer l'exécution du programme. Ici, seul `main` appelle d'autres fonctions

- ▶ Pour aller plus loin...
- ▶ Des outils peuvent analyser la sortie de gprof pour produire une représentation du graphe d'appel
- ▶ Essayez `python optim_base/utis/gprof2dot.py`
- ▶ La sortie générée peut être visualisée par `xdot` ou <https://github.com/jrfonseca/xdot.py> (fourni dans `optim_base/utis/gprof2dot.py`):



- ▶ Gprof permet également d'extraire le temps passé ligne par ligne
- ▶ Compilez `ju.c` avec les options `-pg` et `-g`
- ▶ Exécutez le programme
- ▶ Analysez la trace produite avec `gprof -l`
- ▶ Dans la sortie, les noms de fonctions sont remplacés par des numéros de ligne dans des fichiers sources

# TP 2 : Sortie de gprof

% time	cumulative seconds	self seconds	calls	self Ts/call	total Ts/call	name
64.71	40.34	40.34				evolution (jv.c:68 @ 80487c1)
13.01	48.45	8.11				evolution (jv.c:63 @ 80487f9)
6.50	52.51	4.05				faffiche (jv.c:128 @ 8048942)
3.37	54.60	2.10				evolution (jv.c:74 @ 804882d)
1.82	55.74	1.14				evolution (jv.c:76 @ 804883c)
1.73	56.82	1.08				faffiche (jv.c:127 @ 8048971)
1.48	57.74	0.92				evolution (jv.c:79 @ 804885d)
1.40	58.61	0.87				evolution (jv.c:62 @ 8048808)
1.27	59.40	0.79				evolution (jv.c:82 @ 804886f)
0.76	59.88	0.47				main (jv.c:151 @ 8048a14)
0.74	60.34	0.46				evolution (jv.c:63 @ 80487b6)
0.63	60.73	0.39				evolution (jv.c:62 @ 80487ab)
0.59	61.09	0.37				evolution (jv.c:71 @ 8048817)
0.55	61.44	0.34				evolution (jv.c:54 @ 804887f)
0.47	61.73	0.29				main (jv.c:149 @ 8048a3e)
0.32	61.93	0.20				evolution (jv.c:57 @ 80487a4)
0.25	62.09	0.15				faffiche (jv.c:127 @ 8048939)
0.24	62.23	0.15				evolution (jv.c:80 @ 804886d)

- ▶ Examinons la ligne en question :

```
nbvoisins+=g1[ (( xv+maxi ) %maxi ) *maxj+( yv+maxj )  
               %maxj ] ;
```

- ▶ Elle contient des opérations arithmétiques et une copie mémoire

- ▶ Mesure du temps passé en contexte utilisateur uniquement
- ▶ Surcoût en temps lié aux interruptions régulières
- ▶ Le profiling nécessite une recompilation  $\Rightarrow$  pas de profiling des bibliothèques externes non prévues pour le profiling.

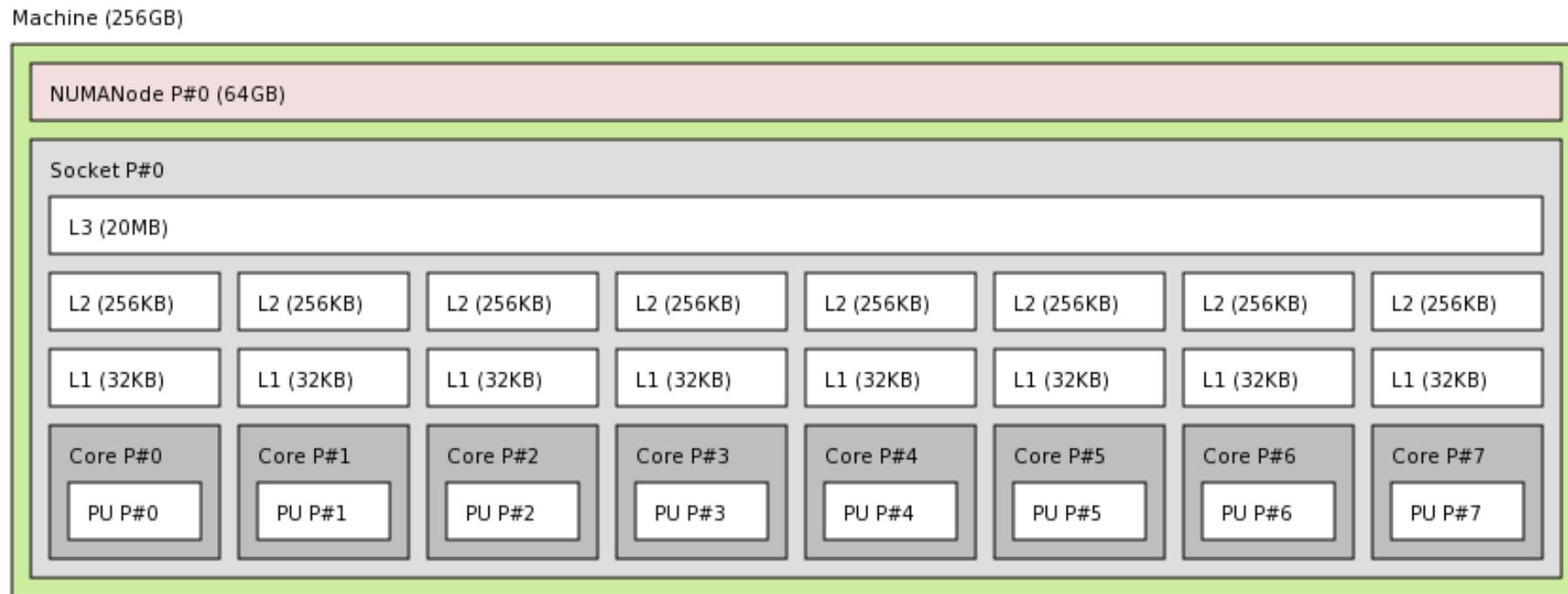
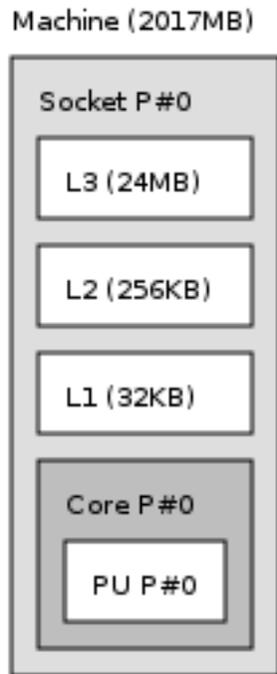
- ▶ Exemple de compilation séparée : fonction évolution est recompilée sans -pg, le graphe d'appels est incomplet :

granularity: each sample hit covers 4 byte(s) for 0.02% of 62.35 seconds

index	% time	self	children	called	name
[1]	89.2	55.59	0.00		<spontaneous> evolution [1]
-----					
[2]	10.8	0.81	5.95		<spontaneous> main [2]
		5.95	0.00	4/4	faffiche [3]
-----					
[3]	9.5	5.95	0.00	4/4	main [2]
		5.95	0.00	4	faffiche [3]
-----					

- ▶ Gprof est un bon **outil de mesure** pour synthétiser des informations quantitatives sur un programme
- ▶ Présent dans de nombreuses distributions Linux
- ▶ Gprof fait partie des premiers outils à utiliser
- ▶ Par contre, si les performances ne sont pas satisfaisantes, il nous faut trouver par nous même les pistes d'amélioration
- ▶ Certains outils mesurent plus précisément le comportement du programme et éclairent sur des pistes à utiliser

- ▶ L'architecture des processeurs actuels est représentée ici :



Machine mono-coeur

Machine multi-coeur

## ► Utilité du cache

- Le cache est une mémoire rapide de petite taille
- Au fur et à mesure de leur utilisation en mémoire, les données sont placées dans les différents niveaux de cache par le matériel
- Le cache étant de taille limitée, il est nécessaire fréquemment de remplacer les données qui s'y trouvent par d'autres données
- Il existe différents algorithmes qui associent une donnée (@ mémoire) et son emplacement dans le cache

- ▶ Le nombre de cycles nécessaires à l'obtention d'une donnée dépend de son emplacement :

Level	Hit
1	5
2	10
3	40-65
Mémoire	60ns (150 / 200 cycles)

Source <http://software.intel.com/en-us/forums/topic/287236>

- ▶ Notez que pour les machines multi-processeurs, les délais dépendent en plus du processeur rattaché à la mémoire correspondante

- ▶ On gagnera donc à utiliser des données qui se trouvent dans le cache
- ▶ Un défaut de cache, le fait d'utiliser une donnée qui ne se trouve pas dans le cache, est coûteux en termes de performances.
- ▶ Les défauts de cache sont nombreux et inévitables, on va néanmoins chercher à les minimiser
- ▶ Il existe 2 méthodes de mesure

- ▶ Mesure basées sur les événements matériels :
  - Les processeurs sont munis d'un grand nombre de compteurs
  - La bibliothèque PAPI (Performance Application Programming Interface) donne accès à ces compteurs de manière portable
  - PAPI est utilisée par d'autres outils comme TAU
- ▶ Mesure basée sur des simulateurs
  - Le simulateur valgrind peut accueillir un simulateur de cache à 3 niveaux

- ▶ Sur la machine du centre de calcul, nous utilisons **valgrind + cachegrind** :

```
david@hpc$ valgrind --tool=cachegrind optim_base/jv/jv.short
==5207== Cachegrind, a cache and branch-prediction profiler
==5207== Copyright (C) 2002-2011, and GNU GPL'd, by Nicholas Nethercote et al.
==5207== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==5207== Command: /usr/local/atelier/jv/jv.short
==5207==
--5207-- warning: L3 cache found, using its data for the LL simulation.
```

Sur la machine support d'exécution

Last-Level (pour le modèle de cache)

==5207==

==5207== I refs: 76,674,125,568



Cache d'instruction : L1

==5207== I1 misses: 1,026

==5207== LLi misses: 1,022

==5207== I1 miss rate: 0.00%

==5207== LLi miss rate: 0.00%

Caches de données : L1 et LL.

Où est L2 ?

==5207==

==5207== D refs: 41,502,666,201 (37,263,848,578 rd + 4,238,817,623 wr)

==5207== D1 misses: 20,354,629 ( 10,979,553 rd + 9,375,076 wr)

==5207== LLd misses: 14,064,843 ( 4,689,848 rd + 9,374,995 wr)

==5207== D1 miss rate: 0.0% ( 0.0% + 0.2% )

==5207== LLd miss rate: 0.0% ( 0.0% + 0.2% )

==5207==

==5207== LL refs: 20,355,655 ( 10,980,579 rd + 9,375,076 wr)

==5207== LL misses: 14,065,865 ( 4,690,870 rd + 9,374,995 wr)

==5207== LL miss rate: 0.0% ( 0.0% + 0.2% )

En plus de la sortie texte, un fichier de profiling est analysable par `cg_annotate` :

```
rdavid@jdevt7b:~/Jeuvie$ cg_annotate cachegrind.out.5314
```

```
-----  
I1 cache:      32768 B, 64 B, 4-way associative  
D1 cache:      32768 B, 64 B, 8-way associative  
LL cache:      25165824 B, 64 B, 24-way associative  
Command:       ./jv.short  
Data file:     cachegrind.out.5314  
Events recorded: lr llmr llmr Dr D1mr DLmr Dw D1mw DLmw  
Events shown:  lr llmr llmr Dr D1mr DLmr Dw D1mw DLmw  
Event sort order: lr llmr llmr Dr D1mr DLmr Dw D1mw DLmw  
Thresholds:    0.1 100 100 100 100 100 100 100 100  
Include dirs:  
User annotated:  
Auto-annotation: off
```

```
-----  
      lr llmr llmr      Dr      D1mr      DLmr      Dw      D1mw      DLmw  
-----  
38,983,423,216  910   906 18,559,529,660 1,564,482 1,564,221 2,333,273,351 6,250,230 6,250,217 PROGRAM TOTALS
```

```
-----  
      lr llmr llmr      Dr      D1mr      DLmr      Dw      D1mw      DLmw  file:function  
-----  
38,441,809,784   5    5 15,259,682,959 1,562,503 1,562,503  601,000,003 1,562,501 1,562,501 /home/rdavid/Jeuvie/jv.c:evolution  
4,100,012,710   3    3 1,596,779,144      0          0 1,000,003,100      0          0 /build/buildd/eglibc-2.15/stdlib/random_r.c:random_  
r  
2,300,000,000   2    2   700,000,000      0          0  500,000,000      0          0 /build/buildd/eglibc-2.15/stdlib/random.c:random  
1,607,000,074   6    6   503,000,018      1          1  201,000,029 1,562,501 1,562,501 /home/rdavid/Jeuvie/jv.c:main  
400,000,778     1    1   400,000,778      0          0      0          0      0 /build/buildd/eglibc-2.15/string/./sysdeps/i386/i6  
86/multiarch/strcat.S:???  
100,000,269    23   20  100,000,128      12         11      63          1          1  ???:???
```

Les événements remontés sont :

Ir : I cache reads (ie. instructions executed)

I1mr: I1 cache read misses

I2mr: L2 cache instruction read misses

**Dr : D cache reads (ie. memory reads)**

D1mr: D1 cache read misses

**D2mr: L2 cache data read misses**

**Dw : D cache writes (ie. memory writes)**

D1mw: D1 cache write misses

**D2mw: L2 cache data write misses**

Il peut être bien plus intéressant d'avoir une remontée d'informations au niveau ligne de code et non par fonction :

```
cg_annotate --auto=yes cachegrind.out.5314
```

nous apprend par exemple que l'instruction :

```
nbvoisins+=g1[((xv+maxi)%maxi)*maxj+(yv+maxj)  
%maxj];
```

est celle qui génère les défauts de cache du programme.

Est-ce logique ?

- ▶ Performance ? What is it all about ?
- ▶ Mesure
- ▶ Optimisation : CPU
- ▶ Optimisation : mémoire
- ▶ Conclusion

- ▶ Pour traiter de 2 techniques d'optimisation CPU, nous travaillons sur une version à peine modifiée des Stream Benchmarks  
<http://www.cs.virginia.edu/stream/ref.html>
- ▶ Ce benchmark réalise des opérations très simples :
  - addition de tableaux
  - copie
  - triad :  $a = b + \text{coefficient} * c$
- ▶ Le bench affiche un résultat en MB/s, que nous allons améliorer

- ▶ Quelles sont (quelques unes) des optimisations CPU réalisées par les compilateurs ?  
Source <http://gcc.gnu.org/onlinedocs/4.8.1/>
  - Sortie des invariants de boucle
  - Réduction de variables en examinant les variables induites (ex : indice et adresse mémoire)
  - Déroutage de boucles ⇒ Réduction coût de gestion
  - Vectorisation ⇒ Réduction temps passé
  - Inlining ⇒ Réduction temps passé dans la gestion des appels

- ▶ Nous allons augmenter le ratio d'instructions utiles :
  - En augmentant relativement la longueur du corps de boucle
  - En diminuant le nombre de tours de boucle

- ▶ Faites une copie du fichier

`optim_base/atelier/Stream/stream.c`

sur votre compte, et ajoutez une directive de compilation `TUNED_ADD` pour proposer une version « fonction » du code contenu dans la ligne 334-335

- ▶ Voici un exemple de boucle déroulée de taille 4 :

```
for (j=0; j<STREAM_ARRAY_SIZE; j=j+4)
{
    a[j] = b[j]+scalar*c[j];
    a[j+1] = b[j+1]+scalar*c[j+1];
    a[j+2] = b[j+2]+scalar*c[j+2];
    a[j+3] = b[j+3]+scalar*c[j+3];
}
```

- ▶ Quelle amélioration de performance observez-vous ?

- ▶ Avec le compilateur Gcc, pour dérouler automatiquement les boucles dans tout le code, utiliser l'option à la ligne de commande :
- ▶ Pour dérouler spécifiquement **une** boucle, on ajoute des instructions (pragma) à destination du compilateur dans le code :

```
-funroll-loops
```

```
#pragma GCC push_options  
#pragma GCC optimize ("unroll-loops")  
définition de fonction  
#pragma GCC pop_options
```

- ▶ Depuis une dizaine d'années, les processeurs disposent de petites unités vectorielles (stockage + calcul) effectuant diverses opérations mathématiques, motivés par les besoins du traitement d'images. 5<sup>ème</sup> version à ce jour <http://sseplus.sourceforge.net/fntable.html>
- ▶ Ces instructions nécessitent l'emploi de types spécifiques, décrits dans des fichiers spécifiques
- ▶ Les fonctions correspondant aux opérateurs arithmétiques sont des fonctions intrinsèques connues par le compilateur... si on le lui demande

- ▶ SSE : 1ers jeux d'instructions vectorielles, accessible sur Intel et AMD
- ▶ AVX : Jeux d'instructions introduit avec le Sandy Bridge, disponible également sur AMD Bulldozer, visant à étendre le SSE
  - Vecteurs de 256 ou 512 bits
  - Apparition d'opérateurs non destructifs comme  $c=a+b$
- ▶ Dans les exemples, nous étudierons des cas très simples issus de SSE 1

- ▶ Exemple de type de données :  
`__m128` : un vecteur de 128 bits  
4 flottants ou 2 doubles
- ▶ Exemple d'opération :  
`_mm_add_ps` : addition terme à terme sur des  
éléments flottants de 4 octets
- ▶ L'opération arithmétique effectuée en une fois les 4  
additions  $\Rightarrow$  gain de temps CPU

- ▶ Dans le code, ajouter l'inclusion de l'en-tête spécifique :  

```
#include <xmmintrin.h>
```
- ▶ Pour que les types et les fonctions soient activés dans le compilateur, ajouter les options `-msse`, jusqu'à `-msse4.2` si le processeur le supporte
- ▶ `-msse` suffit pour nos TP
- ▶ Pour vérifier le niveau de sse pris en charge par le processeur, regardez la sortie de **`cat /proc/cpuinfo`**
- ▶ Instruction assembleur correspondant : **`cquid`**

## ▶ Exemple de code : addition de 2 vecteurs

```
#include <xmmintrin.h>
```

```
__m128 * mm_c= (__m128 *) c; // c,b,a les tableaux
```

**d'origine**

```
__m128 * mm_b= (__m128 *) b;
```

```
__m128 * mm_a= (__m128 *) a;
```

```
for (j=0; j<STREAM_ARRAY_SIZE/4; j++)
```

```
{
```

```
*mm_c = _mm_add_ps(*mm_a,*mm_b);
```

```
// Do some pointer arithmetic on the mm types
```

```
mm_c ++; // Fera avancer le pointeur de 4 octets
```

```
mm_b ++;
```

```
mm_a ++;
```

## ▶ Intégrez ces modifications dans **stream.c** et mesurez le résultat obtenu

- ▶ Résultats obtenus
- ▶ L'exécutable `optim_base/Stream/stream` est la version de base, compilée avec gcc, travaillant sur des float
- ▶ L'exécutable `optim_base/Stream/stream_sse` est la version où les 4 kernels ont été optimisés avec des instructions sse (résultats sur hpc-login) :

Version	Copy	Scale	Add	Triad
Base	2124	2386	2705	3377
Sse	5045	4985	6627	5284

↙ Mb/s

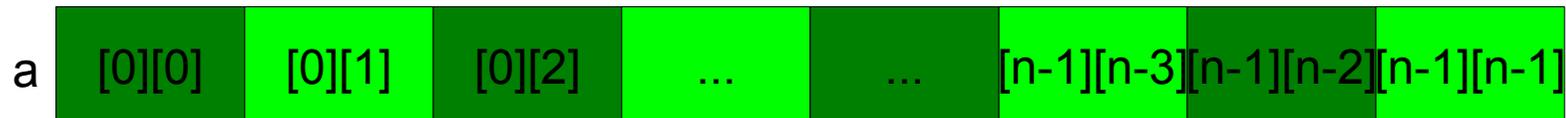
- ▶ Avec le compilateur gcc :
  - `-ftree-vectorize`
  - inclus dans le niveau d'optimisation `-O3`

- ▶ Performance ? What is it all about ?
- ▶ Measure
- ▶ Optimisation : CPU
- ▶ Optimisation : mémoire
- ▶ Conclusion

- ▶ Pour optimiser la mémoire, il faut profiter des données tant qu'elles sont dans le cache.
- ▶ Pour cela, on essaye de profiter de 2 principes de localité :
  - Localité spatiale : travailler sur des données proches en mémoire.  
En effet, proches en mémoire, elles seront vraisemblablement présentes dans le cache en même temps

- Localité temporelle : utiliser les données à plusieurs reprises de manière rapprochée dans le temps  
Cela permet de *rentabiliser* le coût de chargement des données
- ▶ Pour bénéficier de ces caractéristiques, il est parfois nécessaire de restructurer en profondeur les structures mémoire de ses codes.
- ▶ Le travail est comparable à celui qu'on pourrait effectuer lors du portage du code sur un GPU, où les performances sont largement dépendantes des caractéristiques des accès mémoire

- ▶ Ordre des éléments en mémoire pour un tableau 2D en C



- ▶ en C, quand on écrit  $a[i][j]$ , on accède à l'élément d'adresse :

$$a + i \times \text{dim}_y + j$$

Fonction d'accès

- ▶ C'est une convention du langage, d'autres langages procèdent autrement (Fortran)
- ▶ Dans tous les cas, nous devons nous rapprocher dans nos codes de l'ordre d'accès mémoire

- ▶ Dans le fichier `optim_base/cache/t.c`, nous reproduisons pour des tableaux de taille variable la fonction d'accès vue précédemment :
- ▶ Nous allouons une zone contigüe en mémoire :  

```
t=malloc(nb_elem*sizeof(TYPE));
```
- ▶ Nous accédons à cette zone par un adressage 2D :  

```
#define ADDR(i,j,dim) (i*dim+j)
```
- ▶ Correspond exactement à ce qu'aurait fait le compilateur sur un tableau dont il connaît la taille

- ▶ Nous réalisons un parcours du tableau dans 2 fonctions différentes qui produisent le même résultat

```
▶ for (i=0 ; i < dim ; i++)  
    for (j=0 ; j < dim ; j++) ← Fonction parcours_ok  
    {  
        t[ADDR(i, j, dim)] = i+j;  
    }
```

- ▶ Quelle est l'adresse de deux accès consécutifs ?

- ▶ Instrumentez le code et lisez les traces avec gprof
- ▶ Nous lançons le code avec valgrind  
**valgrind -tool=cachegrind ./t 20** (long)
- ▶ Le résultat fait apparaître un nombre de défauts de cache relativement élevés : 50% des accès se traduisent par un cache miss

Allocation 419430400 elements

==2020==

```
==2020== I   refs:      12,583,478,066
==2020== I1  misses:           835
==2020== LLi misses:           834
==2020== I1  miss rate:      0.00%
==2020== LLi miss rate:      0.00%
```

==2020==

```
==2020== D   refs:      8,388,916,188 (7,549,999,949 rd + 838,916,239 wr)
==2020== D1  misses:      445,647,270 (      1,958 rd + 445,645,312 wr)
==2020== LLd misses:      445,647,270 (      1,958 rd + 445,645,312 wr)
==2020== D1  miss rate:      5.3% (      0.0% +      53.1% )
==2020== LLd miss rate:      5.3% (      0.0% +      53.1% )
```

==2020==

```
==2020== LL  refs:      445,648,105 (      2,793 rd + 445,645,312 wr)
==2020== LL  misses:      445,648,104 (      2,792 rd + 445,645,312 wr)
==2020== LL  miss rate:      2.1% (      0.0% +      53.1% )
```

- ▶ Avec `cg_annotate --auto=yes cachegrind.out.[pid]`, on se rend bien compte de l'endroit où se produisent les défauts de cache.
- ▶ Est-ce logique ?
- ▶ Quelle est la différence principale entre les fonctions `parcours_ok` et `parcours_ko` ?

- ▶ Performance ? What is it all about ?
- ▶ Measure
- ▶ Optimisation : CPU
- ▶ Optimisation : mémoire
- ▶ Conclusion

- ▶ La boîte à outils pour l'analyse des performances est vaste... mais peut s'utiliser pas à pas
- ▶ L'optimisation de code dé-structure le code initial
  - Pensez à conserver l'interface et à documenter
  - Certaines optimisations rendent le code dépendant de l'architecture, prévoyez une roue de secours
  - Le compilateur ne fait pas tout
- ▶ Autre piste : les optimisations algorithmiques liées à la connaissance du domaine (symétries, ...)