Micro-benchmarking de précision

Hadrien Grasland

2020-03-04

Optimisation guidée par benchmark

1. **Identifier** un problème de performances
2. **Profiler** pour localiser les composants critiques
3. **Construire** des *benchmarks* de ces composants
4. **Utiliser** ces *benchmarks* pour améliorer leur performance
5. **Vérifier** l'impact sur le système global

Exemple

1. La reconstruction ATLAS est trop lente pour le HL-LHC
2. Grand rôle de la trajectographie, en particulier :
    1. Propagation Runge-Kutta
    2. Navigation dans la géométrie
    3. Filtre de Kalman
3. Des benchmarks de ces composants ont été isolés
4. Ils facilitent le travail sur leur performance
5. L'effet sur une reconstruction complète sera ensuite vérifié

Pourquoi utiliser des benchmarks ?

- Compilent et s'exécutent **rapidement**
  - Cycle de développement efficace
- Accès à des **outils** inutilisables sur gros programme
  - MAQAO, cachegrind, examen de l'asembleur…
- Facilitent la détection & l'analyse de **régressions**
  - …voire permettent leur automatisation
- Un problème simplifié stimule la **créativité**

Quel prix à payer ?

- On **porte des oeillères**, au risque de…
  - …s'acharner quand l'impact est devenu négligeable
  - …manquer une optimisation de plus haut niveau
- Le benchmark peut **ne pas être représentatif**
- Compilateur & matériel peuvent **sur-optimiser**
  - Niveau de performances inaccessible en conditions réelles
- Si on gère bien ces risques, l'impact reste positif

Au programme

- Aujourd'hui, on va parler de…
  - Biais d'origine logicielle (OS, compilateur…)
  - Imprécisions dans la mesure de temps
- On n'évoquera que brièvement les…
  - Biais d'origine matérielle (caches, ILP…)
  - Biais humain d'écriture du benchmark

Démarche

- Des exemples simples, voire simplistes ?
  - Moins d'explications de code, plus d'analyse
  - Suffisant pour exposer de nombreux problèmes
  - Souvent, le goulot d'étranglement *est* simple
  - Utile pour analyser toutes sortes de lieux communs
    - « La racine carrée, c'est lent »
    - « Les branches nuisent aux performances »

Limiter le biais logiciel

Mon Premier Benchmark™

```cpp
#include <chrono>
#include <cstdlib>
#include <iostream>


int main() {
    using Clock = std::chrono::steady_clock;
    using NanoSecs = std::chrono::nanoseconds;

    auto start = Clock::now();

    std::rand();  // Code étudié

    NanoSecs ns = Clock::now() - start;
    std::cout << "T=" << ns.count() << "ns" << std::endl;

    return 0;
}
```

Ex. d'utilisation :

- Je fais du Monte Carlo
- Je veux savoir combien le RNG système produit de nombres par seconde

Quelques exécutions

```
$ g++ -std=c++11 -O3 ex1a.cpp
$ ./a.out
T=1587ns
$ ./a.out
T=1734ns
$ ./a.out
T=1757ns
$ ./a.out
T=1707ns
$ ./a.out
T=1684ns
$ ./a.out
T=1807ns
```

Un résultat assez **reproductible**… Mais est-il **correct** ?

Autre protocole d'exécution

```
$ for i in {1..20}; do ./a.out; done
T=1751ns
T=1404ns
T=735ns
T=723ns
T=797ns
T=706ns
T=712ns
T=1500ns
T=906ns
T=1520ns
T=1440ns
T=1108ns
T=1086ns
T=982ns
T=1054ns
T=723ns
T=686ns
T=661ns
T=770ns
T=736ns
```

Grandes fluctuations !

>2x plus petit qu'avant !

SUSPECT !

Un 3e protocole d'exécution

```cpp
#include <chrono>
#include <cstdlib>
#include <iostream>


#ifndef NUM_RUNS
#define NUM_RUNS 20
#endif

using NanoSecs = std::chrono::nanoseconds;

NanoSecs bench() {
    using Clock = std::chrono::steady_clock;
    auto start = Clock::now();

    std::rand();  // Code étudié

    return Clock::now() - start;
}

int main() {
    std::cout << "T_run(ns)" << std::endl;

    for (size_t i = 0; i < NUM_RUNS; ++i) {
        double nanosecs = bench().count();
        std::cout << nanosecs << std::endl;
    }

    return 0;
}
```

Nb d'exécutions configurable

Equivalent au programme précédent

Pour s'abstraire des effets du shell, on boucle *dans le processus*

Autre résultat

```
$ g++ -std=c++11 -O3 -DNUM_RUNS=15 ex1b.cpp
$ ./a.out
T_run(ns)
2149
164
97
80
120
74
71
74
67
77
77
74
74
73
77
```

« Chauffe » des caches,
mécanismes paresseux de l'OS...

« Vrai » état d'équilibre ?
Un seul moyen de savoir...

Autres NUM_RUNS

```
$ g++ -std=c++11 -O3 -DNUM_RUNS=100 ex1b.cpp && ./a.out
   [ … blabla … ]
81
80
83

$ g++ -std=c++11 -O3 -DNUM_RUNS=1000 ex1b.cpp && ./a.out
   [ … blabla … ]
68
55
58

$ g++ -std=c++11 -O3 -DNUM_RUNS=10000 ex1b.cpp && ./a.out
   [ … blabla … ]
47
48
44

$ g++ -std=c++11 -O3 -DNUM_RUNS=100000 ex1b.cpp && ./a.out
   [ … blabla … ]
26
26
25
```

On en était encore loin !

Certains mécanismes de « chauffe » sont plus lents que d'autres...

Autres NUM_RUNS

```
$ g++ -std=c++11 -O3 -DNUM_RUNS=10000000 ex1b.cpp && ./a.out
   [ … blabla … ]
38
32
32
36
32
37
28
34
30
33
27
26
27
25
27
26
28
27
    [ … blabla … ]
```

Stable à un facteur ~2x près après ~1s

Pas d'amélioration de stabilité ensuite

Mais mesure-t'on std::rand()… ou le reste de la boucle ?

Test de linéarité

```
 8 #ifndef NUM_RUNS
 9 #define NUM_RUNS 20
10 #endif
11
12 #ifndef NUM_ITERS
13 #define NUM_ITERS 1
14 #endif
15
16 using NanoSecs = std::chrono::nanoseconds;
17
18 NanoSecs bench_run() {
19     using Clock = std::chrono::steady_clock;
20     auto start = Clock::now();
21
22     for (size_t iter = 0; iter < NUM_ITERS; ++iter) {
23         std::rand();  // Code étudié
24     }
25
26     return Clock::now() - start;
27 }
28
29 int main() {
30     size_t col1_width = 15;
31     std::cout << std::left;
32     std::cout << std::setw(col1_width) << "T_run(µs)"
33               << "T_iter,avg(ns)" << std::endl;
34
35     for (size_t i = 0; i < NUM_RUNS; ++i) {
36         double t_run = bench_run().count();
37         double t_iter = t_run / NUM_ITERS;
38         std::cout << std::setw(col1_width-1) << t_run / 1000
39                   << t_iter << std::endl;
40     }
41
42     return 0;
43 }
```

Boucle sur nouveau NUM_ITERS

Affichage à deux colonne :
- Durée totale du benchmark
- Temps moyen d'itération

Augmentons NUM_ITERS…

```
$ g++ -std=c++11 -O3 -DNUM_RUNS=1000000 -DNUM_ITERS=1 ex1c.cpp && ./a.out
   [ … blabla … ]
0.036           36
0.027           27
0.032           32

$ g++ -std=c++11 -O3 -DNUM_RUNS=1000000 -DNUM_ITERS=10 ex1c.cpp && ./a.out
   [ … blabla … ]
0.119           11.9
0.097           9.7
0.099           9.9

$ g++ -std=c++11 -O3 -DNUM_RUNS=1000000 -DNUM_ITERS=100 ex1c.cpp && ./a.out
   [ … blabla … ]
0.733           7.33
0.727           7.27
0.729           7.29

$ g++ -std=c++11 -O3 -DNUM_RUNS=1000000 -DNUM_ITERS=1000 ex1c.cpp && ./a.out
   [ … blabla … ]
6.811           6.811
6.802           6.802
6.797           6.797
```

x3 /3

x7 /1,4

x9 /1,1

…toujours plus haut, toujours plus fort !

```
$ g++ -std=c++11 -O3 -DNUM_RUNS=100000 -DNUM_ITERS=10000 ex1c.cpp && ./a.out
   [ … blabla … ]
67.250          6.7250
67.681          6.7681
67.452          6.7452

$ g++ -std=c++11 -O3 -DNUM_RUNS=10000 -DNUM_ITERS=100000 ex1c.cpp && ./a.out
   [ … blabla … ]
683.244         6.83244
676.905         6.76905
676.306         6.76306

$ g++ -std=c++11 -O3 -DNUM_RUNS=1000 -DNUM_ITERS=1000000 ex1c.cpp && ./a.out
    [ … blabla … ]
6750.66         6.75066
6766.02         6.76602
6820.27         6.82027
```

…bon, $T_{iter,avg}$ est ~**stable**, T_{run} **linéaire** en NUM_ITERS, ça semble OK.

Essayons maintenant de mesurer *autre chose* que std::rand()…

Nouvelle opération benchmark

```
18 NanoSecs bench_run() {
19     using Clock = std::chrono::steady_clock;
20     auto start = Clock::now();
21
22     for (size_t iter = 0; iter < NUM_ITERS; ++iter) {
23         std::sqrt(4.2);  // Code étudié
24     }
25
26     return Clock::now() - start;
27 }
```

Lieu commun du calcul : « La racine carrée est lente, donc à éviter »

Ce genre de savoir millénaire doit toujours être…

- **Vérifié :** Est-ce toujours vrai sur les systèmes actuels ?
- **Quantifié :** Combien de fois plus lent qu'une somme ?

Sauf que...

Décroissant ?

```
$ g++ -std=c++11 -O3 -DNUM_RUNS=1000000 -DNUM_ITERS=1 ex2a.cpp && ./a.out
   [ … blabla … ]
0.032          32
0.033          33
0.035          35

$ g++ -std=c++11 -O3 -DNUM_RUNS=1000000 -DNUM_ITERS=10 ex2a.cpp && ./a.out
   [ … blabla … ]
0.018          1.8
0.018          1.8
0.017          1.7

$ g++ -std=c++11 -O3 -DNUM_RUNS=1000000 -DNUM_ITERS=100 ex2a.cpp && ./a.out
   [ … blabla … ]
0.016          0.16
0.017          0.17
0.017          0.17

$ g++ -std=c++11 -O3 -DNUM_RUNS=1000000 -DNUM_ITERS=1000 ex2a.cpp && ./a.out
   [ … blabla … ]
0.017          0.017
0.017          0.017
0.017          0.017
```

<< 1 cycle processeur !

Explication

```
$ objdump -d a.out

    [ ... blabla ... ]

0000000000401350 <_Z9bench_runv>:
  401350:       53                      push   %rbx
  401351:       e8 4a fd ff ff          callq  4010a0
<_ZNSt6chrono3_V212steady_clock3nowEv@plt>
  401356:       48 89 c3                mov    %rax,%rbx
  401359:       e8 42 fd ff ff          callq  4010a0
<_ZNSt6chrono3_V212steady_clock3nowEv@plt>
  40135e:       48 29 d8                sub    %rbx,%rax
  401361:       5b                      pop    %rbx
  401362:       c3                      retq
  401363:       66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
  40136a:       00 00 00
  40136d:       0f 1f 00                nopl   (%rax)

    [ ... blabla ... ]
```

Mesure d'horloge

Mesure d'horloge

- La valeur de sqrt(4,2) n'est pas utilisée
- Son calcul n'a pas d'effet de bord

→ L'optimiseur a éliminé le calcul !

Cas plus complexe

```
19 NanoSecs bench_run() {
20     using Clock = std::chrono::steady_clock;
21     auto start = Clock::now();
22 
23     for (size_t iter = 0; iter < NUM_ITERS; ++iter) {
24         std::sqrt(std::rand());  // Code étudié
25     }
26 
27     return Clock::now() - start;
28 }
```

On a déjà vu que le compilateur n'élimine pas l'appel à std::rand()

- Manipulation d'un état global ou thread-local → effet de bord

Que va-t'il faire si on calcule une racine carrée à partir de cette valeur ?

- Non trivial : Effet de bord possible via *errno* !

Résultat avec GCC 9.2.1

- std::rand() seul :

```
$ g++ -std=c++11 -O3 -DNUM_RUNS=10000 -DNUM_ITERS=10000 ex1c.cpp && ./a.out
    [ … blabla … ]
67.665          6.7665
67.667          6.7667
67.256          6.7256
```

- std::sqrt(std::rand()) :

```
$ g++ -std=c++11 -O3 -DNUM_RUNS=10000 -DNUM_ITERS=10000 ex2b.cpp && ./a.out
    [ … blabla … ]
70.078          7.0078
69.986          6.9986
69.934          6.9934
```

~1 cycle CPU de plus

Alors, sqrt aussi rapide qu'une addition ?

Dans le doute, cf assembleur

Avant :

```
0000000000401370 <_Z9bench_runv>:
  401370:  push   %rbp
  401371:  push   %rbx
  401372:  mov    $0x2710,%ebx
  401377:  sub    $0x8,%rsp
  40137b:  callq  4010b0 <_ZNSt6chrono3_V212steady_clock3nowEv@plt>
  401380:  mov    %rax,%rbp
  401383:  nopl   0x0(%rax,%rax,1)
  401388:  callq  401040 <rand@plt>
  40138d:  sub    $0x1,%rbx
  401391:  jne    401388 <_Z9bench_runv+0x18>
  401393:  callq  4010b0 <_ZNSt6chrono3_V212steady_clock3nowEv@plt>
  401398:  add    $0x8,%rsp
  40139c:  sub    %rbp,%rax
  40139f:  pop    %rbx
  4013a0:  pop    %rbp
  4013a1:  retq
    [ … pas du code … ]
```

Traduction du nouveau code :

- Convertir la sortie de rand() en double
- Voir si le nombre est négatif
- Si oui, appeler sqrt, sinon ne rien faire

Après :

```
00000000004013a0 <_Z9bench_runv>:
  4013a0:  push   %rbp
  4013a1:  push   %rbx
  4013a2:  mov    $0x2710,%ebx
  4013a7:  sub    $0x8,%rsp
  4013ab:  callq  4010c0 <_ZNSt6chrono3_V212steady_clock3nowEv@plt>
  4013b0:  mov    %rax,%rbp
  4013b3:  nopl   0x0(%rax,%rax,1)
  4013b8:  callq  401040 <rand@plt>
  4013bd:  pxor   %xmm0,%xmm0
  4013c1:  pxor   %xmm1,%xmm1
  4013c5:  cvtsi2sd %eax,%xmm0
  4013c9:  ucomisd %xmm0,%xmm1
  4013cd:  ja     4013e4 <_Z9bench_runv+0x44>
  4013cf:  sub    $0x1,%rbx
  4013d3:  jne    4013b8 <_Z9bench_runv+0x18>
  4013d5:  callq  4010c0 <_ZNSt6chrono3_V212steady_clock3nowEv@plt>
  4013da:  add    $0x8,%rsp
  4013de:  sub    %rbp,%rax
  4013e1:  pop    %rbx
  4013e2:  pop    %rbp
  4013e3:  retq
  4013e4:  callq  401070 <sqrt@plt>
  4013e9:  jmp    4013cf <_Z9bench_runv+0x2f>
  4013eb:  nopl   0x0(%rax,%rax,1)
```

Nouveau code

Nouveau code

Implications

- Ce type de « sur-optimisation » est très pernicieux
  - En apparence, l'opération n'a pas été éliminée
  - En réalité, elle a été *partiellement* éliminée
  - Facile d'en tirer de mauvaises conclusions
  - Souvent difficile de vérifier l'assembleur
- Solution : Du point de vue du compilateur…
  - L'entrée de la fonction doit changer à chaque itération
  - La sortie de la fonction doit faire l'objet d'un effet de bord

Une approche classique

```
20 std::pair<double, NanoSecs> bench_run() {
21     using Clock = std::chrono::steady_clock;
22     auto start = Clock::now();
23
24     double sum = 0.;
25     for (size_t iter = 0; iter < NUM_ITERS; ++iter) {
26         double res = std::sqrt(std::rand());  // Code étudié
27         sum += (2 * int(iter%2) - 1) * res;
28     }
29
30     return std::make_pair(sum, Clock::now() - start);
31 }
32
33 int main() {
34     size_t col1_width = 15;
35     std::cout << std::left;
36     std::cout << std::setw(col1_width) << "Result"
37               << std::setw(col1_width) << "T_run(µs)"
38               << "T_iter,avg(ns)" << std::endl;
39
40     for (size_t i = 0; i < NUM_RUNS; ++i) {
41         auto res_and_time = bench_run();
42         double t_run = res_and_time.second.count();
43         double t_iter = t_run / NUM_ITERS;
44         std::cout << std::setw(col1_width) << res_and_time.first
45                   << std::setw(col1_width-1) << t_run / 1000
46                   << t_iter << std::endl;
47     }
48
49     return 0;
50 }
```

- Entrées aléatoires
- Réduction des sorties

Nouvelle colonne « Sortie ».

Sans intérêt pour nous, mais nécessaire dans cette approche de *benchmark*.

L'élimination de code mort marche *entre fonctions* !

Résultat

```
$ g++ -std=c++11 -O3 -DNUM_RUNS=10000 -DNUM_ITERS=100000 ex2c.cpp && ./a.out
    [ … blabla … ]
3.67738e+06     801.718         8.01718
-448656         797.564         7.97564
-1.57817e+06    797.571         7.97571
```

```
$ objdump -d a.out
    [ … blabla … ]
0000000000401450 <_Z9bench_runv>:
  401450:  push   %rbp
  401451:  push   %rbx
  401452:  xor    %ebx,%ebx
  401454:  sub    $0x18,%rsp
  401458:  callq  4010c0 <_ZNSt6chrono3_V212steady_clock3nowEv@plt>
  40145d:  movq   $0x0,(%rsp)
  401464:
  401465:  mov    %rax,%rbp
  401468:  nopl   0x0(%rax,%rax,1)
  40146f:
  401470:  callq  401040 <rand@plt>
  401475:  pxor   %xmm0,%xmm0
  401479:  pxor   %xmm2,%xmm2
  40147d:  cvtsi2sd %eax,%xmm0
  401481:  ucomisd %xmm0,%xmm2
  401485:  movapd %xmm0,%xmm1
  401489:  sqrtsd %xmm1,%xmm1
  40148d:  ja     4014d3 <_Z9bench_runv+0x83>
  40148f:  mov    %ebx,%eax
  401491:  mov    $0xffffffff,%edx
```

Racine carrée !

```
  401496:  pxor   %xmm0,%xmm0
  40149a:  add    $0x1,%rbx
  40149e:  and    $0x1,%eax
  4014a1:  lea    (%rdx,%rax,2),%eax
  4014a4:  cvtsi2sd %eax,%xmm0
  4014a8:  mulsd  %xmm1,%xmm0
  4014ac:  addsd  (%rsp),%xmm0
  4014b1:  movsd  %xmm0,(%rsp)
  4014b6:  cmp    $0x186a0,%rbx
  4014bd:  jne    401470 <_Z9bench_runv+0x20>
  4014bf:  callq  4010c0 <_ZNSt6chrono3_V212steady_clock3nowEv@plt>
  4014c4:  movsd  (%rsp),%xmm0
  4014c9:  add    $0x18,%rsp
  4014cd:  sub    %rbp,%rax
  4014d0:  pop    %rbx
  4014d1:  pop    %rbp
  4014d2:  retq
  4014d3:  movsd  %xmm1,0x8(%rsp)
  4014d9:  callq  401070 <sqrt@plt>
  4014de:  movsd  0x8(%rsp),%xmm1
  4014e4:  jmp    40148f <_Z9bench_runv+0x3f>
  4014e6:  nopw   %cs:0x0(%rax,%rax,1)
    [ … blabla … ]
```

Limites de cette technique

- L'approche doit être adaptée à chaque problème
  - On ne génére pas des flottants comme des entiers
  - Tout résultat n'a pas une « somme » évidente
  - Il faut trouver un « bon » effets de bord
- Un compilateur sophistiqué pourrait sur-optimiser pour la réduction et le générateur aléatoire utilisé
- Contribution génération aléatoire & réduction difficile à isoler
- Sorties « inutiles » trop facilement éliminées par un collègue

L’approche *inline(never)*…

- Une autre technique ancienne : désactiver l’inlining
  - Théorie : Le compilateur ne peut spécialiser sans inlining
  - Problèmes :
    - Le compilateur sait spécialiser si il y a peu d’appels
    - Le compilateur raisonne sur les interfaces (pure, etc.)
    - Pas de méthode standard pour désactiver l’*inlining*
    - Fichier source séparé ? Obsolète avec la LTO/WPO
- …en un mot, cette approche n’est plus à conseiller

3e approche : détourner l'*inline assembly*

- Basée sur les assembleurs *inline* de type « GCC » :

```
int a=10, b;
asm ("movl %1, %%eax;
      movl %%eax, %0;"
     :"=r"(b)         /* output */
     :"r"(a)          /* input */
     :"%eax"          /* clobbered register */);
```

- Comment en faire une barrière d'optimisation ?
  - On passe un pointeur sur une donnée à l'assembleur inline
  - On déclare qu'on peut lire ou écrire partout dans la cible
  - Le compilateur est, en pratique, forcé de nous croire
- Non portable en théorie, largement portable en pratique

Mise en pratique

```
18 template <typename T>
19 void assume_accessed(T&& value) {
20     __asm__ volatile("" : : "g"(&value) : "memory");
21 }
22
23 NanoSecs bench_run() {
24     using Clock = std::chrono::steady_clock;
25     auto start = Clock::now();
26
27     double x = 4.2;
28     for (size_t iter = 0; iter < NUM_ITERS; ++iter) {
29         assume_accessed(x);
30         double res = std::sqrt(x);  // Code étudié
31         assume_accessed(res);
32     }
33
34     return Clock::now() - start;
35 }
```

Hack encapsulé

« modifie » x

« lit » res

Mesures obtenues

```
$ g++ -std=c++11 -O3 -DNUM_RUNS=20 -DNUM_ITERS=100000000 ex3a.cpp && ./a.out
T_run(µs)     T_iter,avg(ns)
404327        4.04327
390330        3.9033
390454        3.90454
390403        3.90403
390268        3.90268
390501        3.90501
390379        3.90379
390275        3.90275
390356        3.90356
390299        3.90299
390355        3.90355
390304        3.90304
390288        3.90288
390271        3.90271
390281        3.90281
390527        3.90527
390448        3.90448
390667        3.90667
390302        3.90302
390281        3.90281
```

Notez que l'approche « aléatoire + somme » ne donnait qu'un résultat 1,25ns supérieur à la mesure std::rand()

Le parallélisme d'instruction complique l'intérprétation du temps d'exécution d'un code complexe.

(D'ailleurs, ici nous mesurons le nombre de racines carrées par seconde « crête » de la machine, quand elle n'a rien d'autre à faire…)

Vérifions l'assembleur

```
00000000004013d0 <_Z9bench_runv>:
    [ … prologue … ]
  4013df:  callq  4010b0 <_ZNSt6chrono3_V212steady_clock3nowEv@plt>
  4013e4:  lea    0x10(%rsp),%r12
  4013e9:  pxor   %xmm2,%xmm2
  4013ed:  lea    0x18(%rsp),%rbp
  4013f2:  mov    %rax,%r13
  4013f5:  mov    0xc24(%rip),%rax        # 402020 <_IO_stdin_used+0x20>
  4013fc:  mov    %rax,0x10(%rsp)
  401401:  nopl   0x0(%rax)
  401408:  movsd  0x10(%rsp),%xmm0
  40140e:  ucomisd %xmm0,%xmm2
  401412:  movapd %xmm0,%xmm1
  401416:  sqrtsd %xmm1,%xmm1
  40141a:  ja     40143b <_Z9bench_runv+0x6b>
  40141c:  movsd  %xmm1,0x18(%rsp)
  401422:  sub    $0x1,%rbx
  401426:  jne    401408 <_Z9bench_runv+0x38>
  401428:  callq  4010b0 <_ZNSt6chrono3_V212steady_clock3nowEv@plt>
    [ … épilogue … ]
  40143b:  movsd  %xmm1,0x8(%rsp)
  401441:  callq  401060 <sqrt@plt>
  401446:  movsd  0x8(%rsp),%xmm1
  40144c:  movsd  %xmm1,0x18(%rsp)
  401452:  sub    $0x1,%rbx
  401456:  pxor   %xmm2,%xmm2
  40145a:  jne    401408 <_Z9bench_runv+0x38>
  40145c:  jmp    401428 <_Z9bench_runv+0x58>
  40145e:  xchg   %ax,%ax
```

OK, ça marche

« Coeur de boucle » bien plus lisible !

Réduisons le travail manuel…

```
18 template <typename T>
19 void assume_accessed(T&& value) {
20     __asm__ volatile("" : : "g"(&value) : "memory");
21 }
22
23 template <typename Func>
24 void bench_iter(Func&& iteration) {
25     assume_accessed(iteration);
26     auto res = iteration();
27     assume_accessed(res);
28 }
29
30 NanoSecs bench_run() {
31     using Clock = std::chrono::steady_clock;
32     auto start = Clock::now();
33
34     double x = 4.2;
35     for (size_t iter = 0; iter < NUM_ITERS; ++iter) {
36         bench_iter([&x] { return std::sqrt(x); });  // Code étudié
37     }
38
39     return Clock::now() - start;
40 }
```

...et hop, on peut extraire la logique

main.cpp

```cpp
#include <cmath>
#include <iomanip>
#include <ios>
#include <iostream>

#include "benchLib.hpp"


#ifndef NUM_RUNS
#define NUM_RUNS 20
#endif

#ifndef NUM_ITERS
#define NUM_ITERS 1
#endif

int main() {
    // Code étudié
    double x = 4.2;
    auto iteration = [&x] { return std::sqrt(x); };

    size_t col1_width = 15;
    std::cout << std::left;
    std::cout << std::setw(col1_width) << "T_run(µs)"
              << "T_iter,avg(ns)" << std::endl;

    for (size_t i = 0; i < NUM_RUNS; ++i) {
        double t_run = bench_run(iteration, NUM_ITERS).count();
        double t_iter = t_run / NUM_ITERS;
        std::cout << std::setw(col1_width-1) << t_run / 1000
                  << t_iter << std::endl;
    }

    return 0;
}
```

benchLib.hpp

```cpp
#pragma once

#include <chrono>


using NanoSecs = std::chrono::nanoseconds;

template <typename T>
void assume_accessed(T&& value) {
    __asm__ volatile("" : : "g"(&value) : "memory");
}

template <typename Func>
void bench_iter(Func&& iteration) {
    assume_accessed(iteration);
    auto res = iteration();
    assume_accessed(res);
}

template <typename Func>
NanoSecs bench_run(Func&& iteration, size_t num_iters) {
    using Clock = std::chrono::steady_clock;
    auto start = Clock::now();

    for (size_t iter = 0; iter < num_iters; ++iter) {
        bench_iter(iteration);
    }

    return Clock::now() - start;
}
```

Métrologie du temps d’exécution

Où en sommes-nous ?

- Jusqu'ici, on a regardé la stabilité des temps « à l'oeil »
- Il est plus que temps d'introduire de vraies barres d'erreur
  - Sur le temps de « run », un écart-type est vite calculé
  - Mais attention en inférant l'écart type du temps d'itération
    - $\mathrm{Var}(T_{run}) = \mathrm{Var}(\mathrm{Somme}(i, T_{iter,i}))$
    - Indépendance : $\mathrm{Var}(T_{run}) = \mathrm{Somme}(i, \mathrm{Var}(T_{iter,i}))$
    - Même distribution : $\mathrm{Var}(T_{run}) = N_{iter} \times \mathrm{Var}(T_{iter})$
    - Donc *sous ces hypothèses*, $\sigma_{iter} = \sigma_{run} / \sqrt{N_{iter}}$

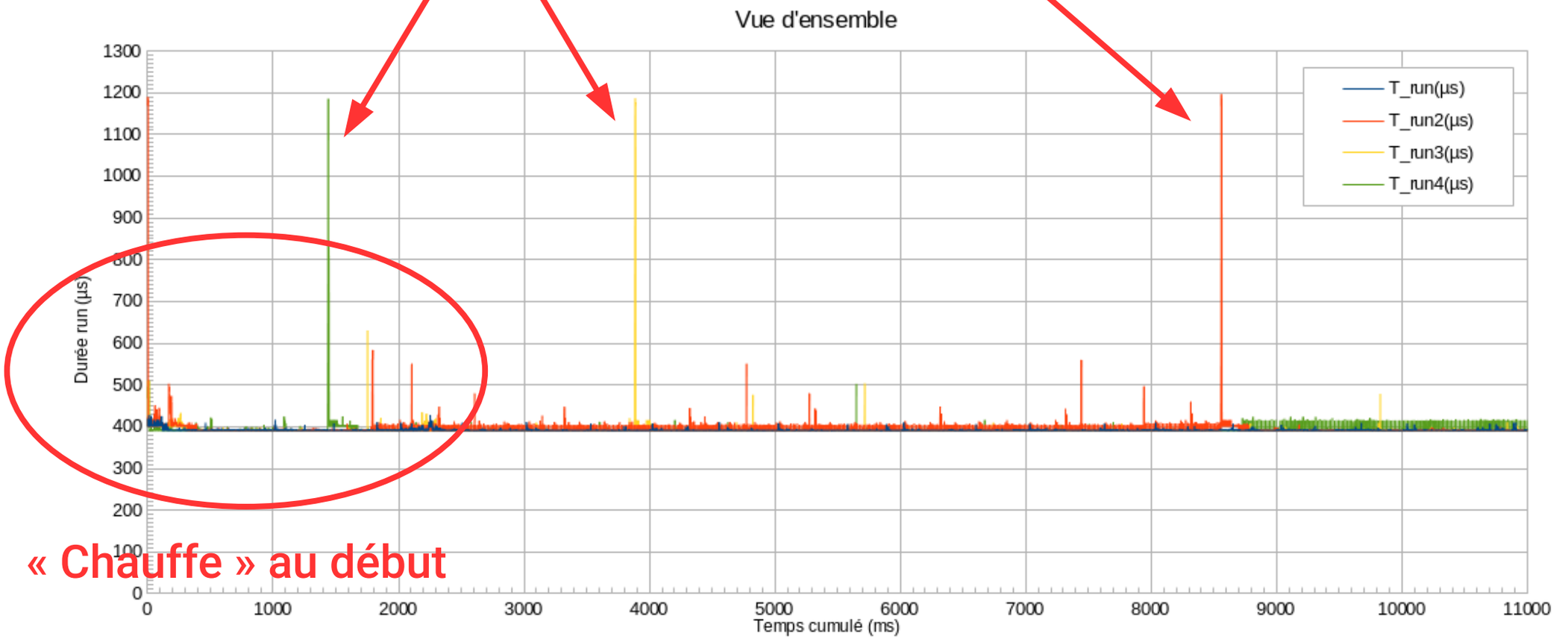
Résultats

```
$ g++ -std=c++11 -O3 -DNUM_RUNS=10000 -DNUM_ITERS=100000 ex4a.cpp && ./a.out
    [ … blabla … ]
389.808         3.89808
390.277         3.90277
389.808         3.89808
390.212         3.90212
389.814         3.89814
389.814         3.89814
389.809         3.89809
389.81          3.8981
389.81          3.8981
394.425         3.94425
389.813         3.89813
========================
T_tot = 3919.15ms
T_run ~ 391.915+/-1493.37µs
T_iter ~ 3.91915+/-4722.46ns
```

Ouch !

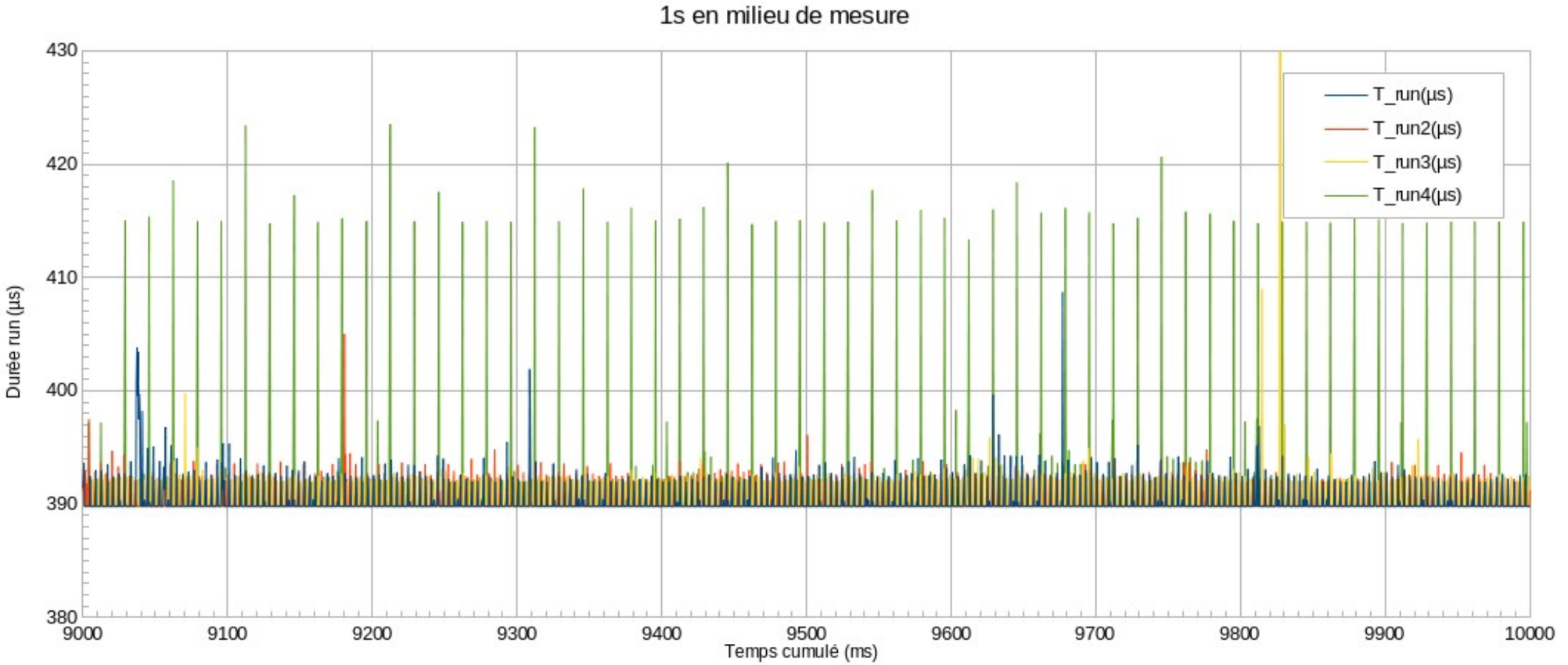
Examinons les temps…

Evénements rares de grande ampleur…

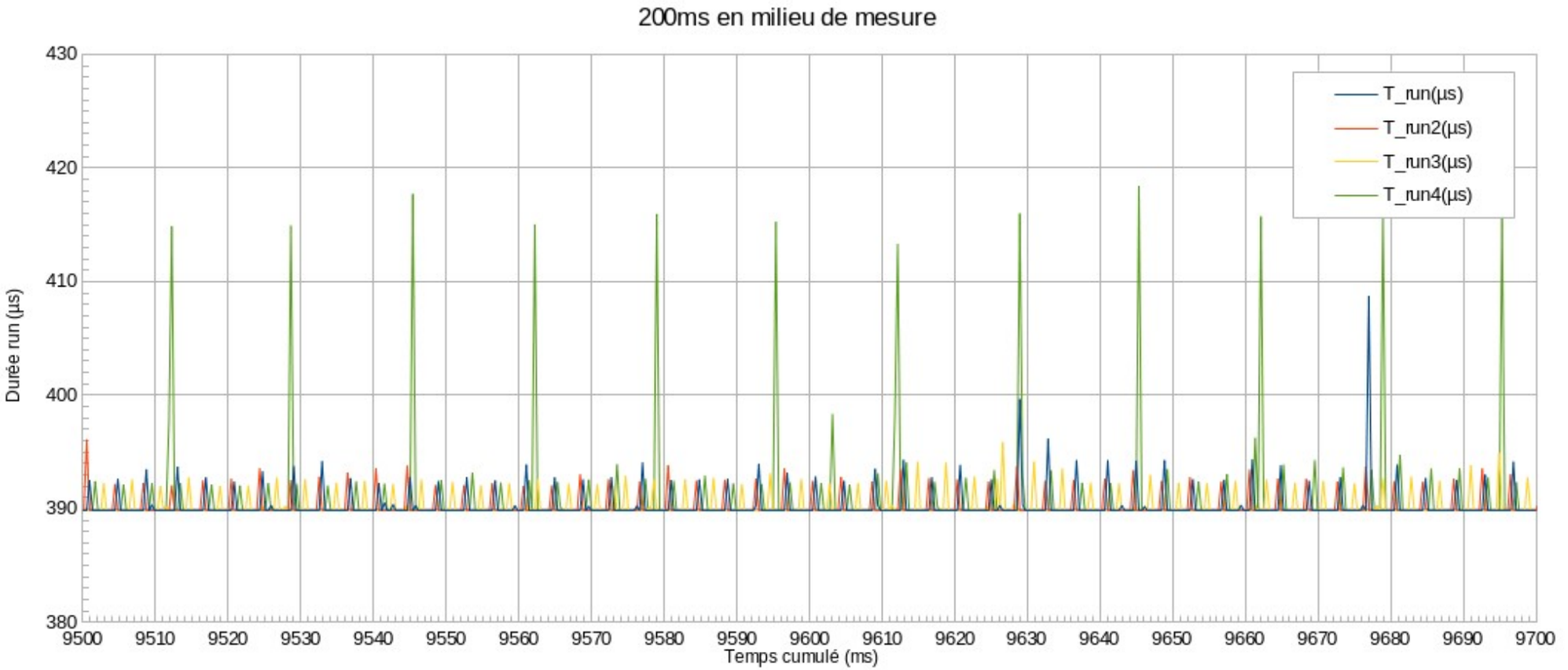


« Chauffe » au début

Zoomons un peu…



...encore un peu



Premières conclusions

- Il faut éjecter la période de chauffe (~1s du début) de l'analyse
  - A la fois bruitée et non reproductible
- Perturbations périodiques de très grande ampleur
  - Ordonnanceur du système d'exploitation + IRQs
  - Pour diminuer leur poids statistique, il faut privilégier un grand nombre de mesures courtes (<<1ms)

Effets de ces changements

Avant :

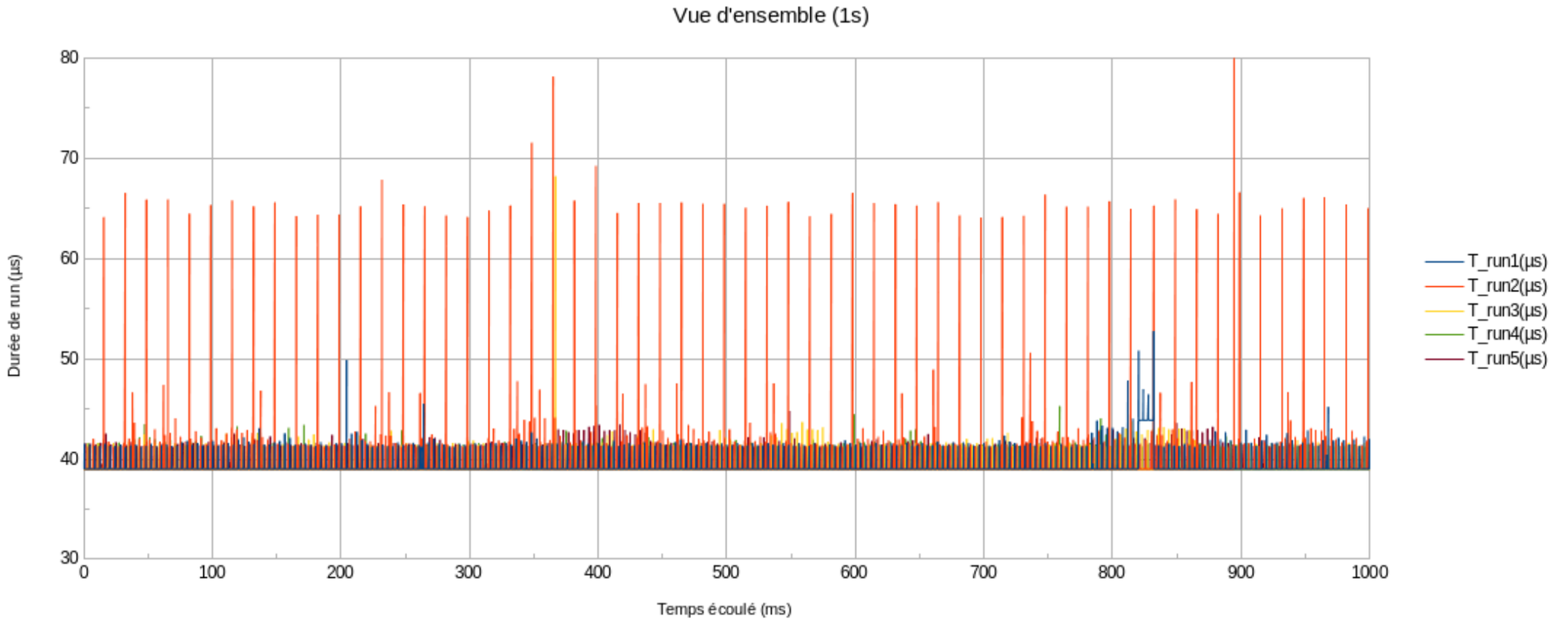
```
    [ … blabla … ]
389.808         3.89808
390.277         3.90277
389.808         3.89808
390.212         3.90212
389.814         3.89814
389.814         3.89814
389.809         3.89809
389.81          3.8981
389.81          3.8981
394.425         3.94425
389.813         3.89813
========================
T_tot = 3919.15ms
T_run ~ 391.915+/-1493.37µs
T_iter ~ 3.91915+/-4722.46ns
```

Après :

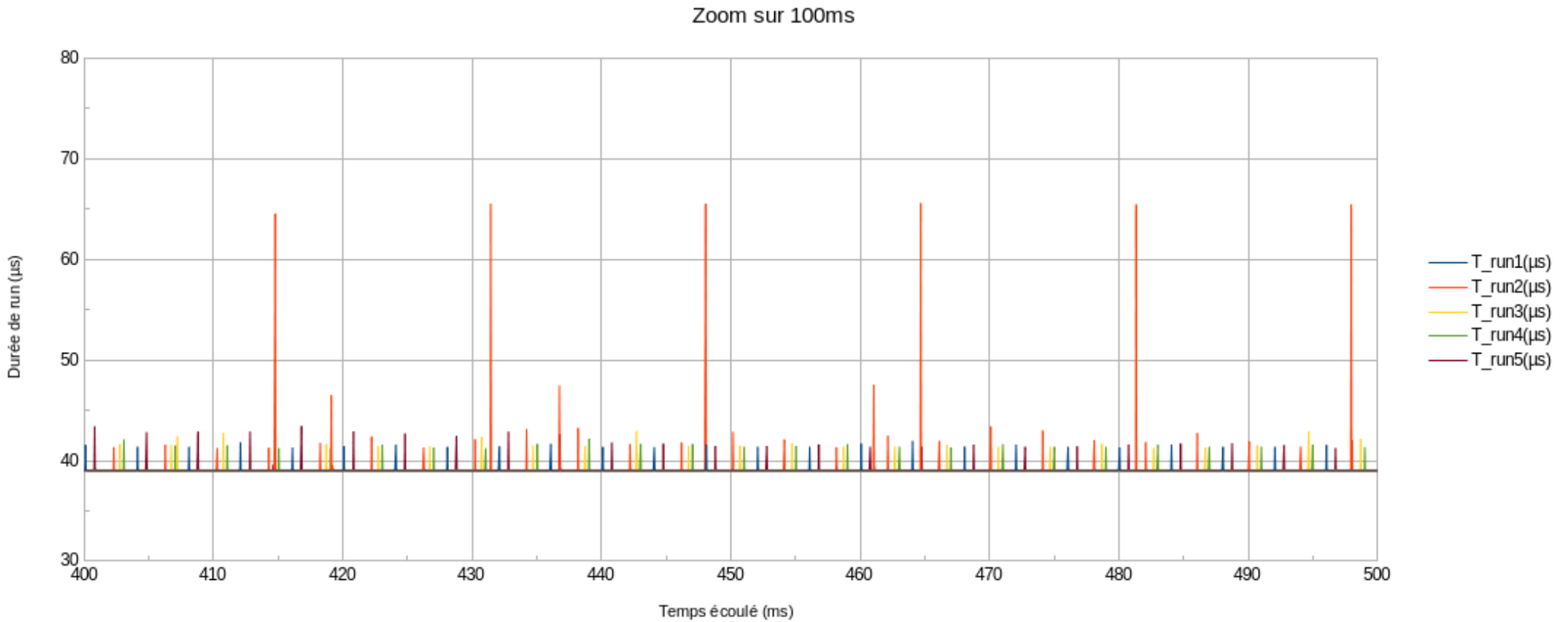
```
    [ … blabla … ]
38.999          3.8999
38.996          3.8996
38.996          3.8996
38.997          3.8997
38.995          3.8995
38.998          3.8998
38.996          3.8996
38.997          3.8997
38.997          3.8997
38.995          3.8995
38.998          3.8998
========================
T_tot = 3902.67ms
T_run ~ 39.0267+/-97.2103µs
T_iter ~ 3.90267+/-972.103ns
```

C’est mieux, mais ce n’est pas encore ça...

Nouvelles mesures, vue d'ensemble

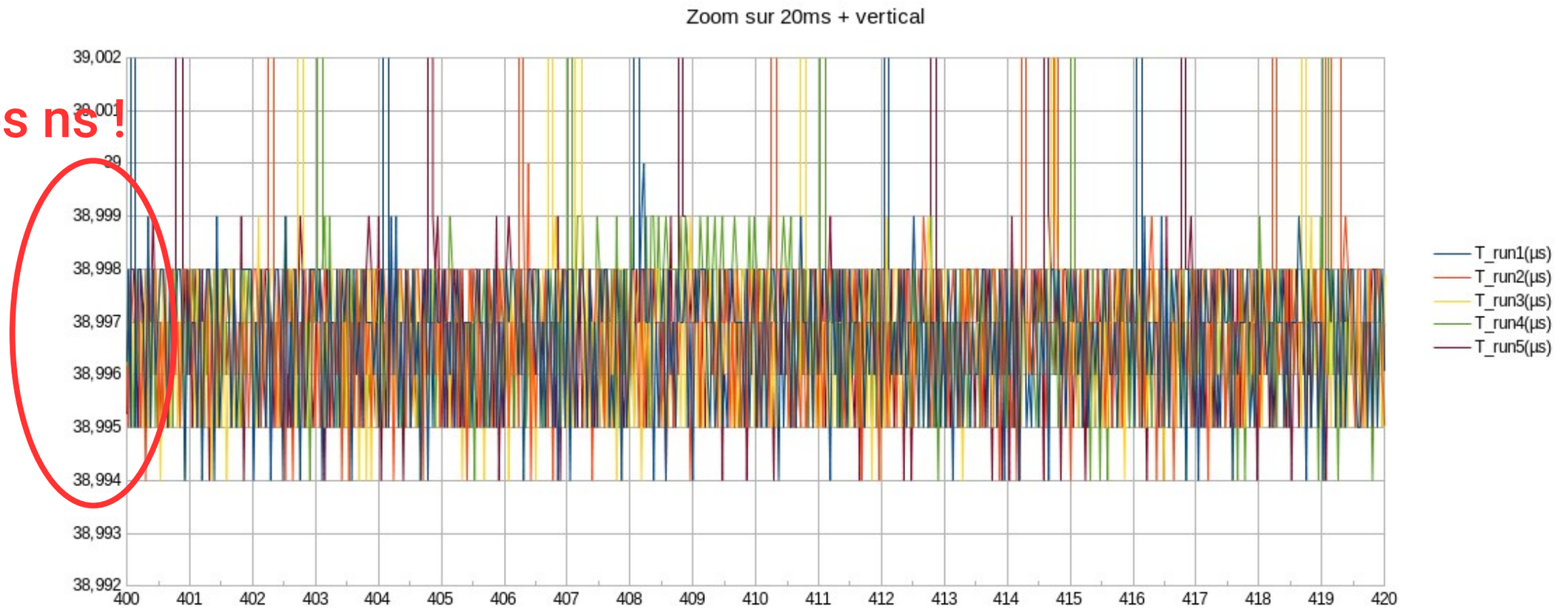


Zoomons un peu…



...encore un peu

quelques ns !



Nouvelles conclusions

- Nous mesurons dans un état stable
- La majorité de nos mesures sont précises à quelques ns
  - …mais la contribution des « pics ordonnanceur » à nos statistiques reste forte, et dissimule cette vraie résolution
- Solution : Utiliser des statistiques robustes aux *outliers*
  - Médiane = Estimateur robuste de la moyenne
  - On peut également dériver un estimateur robuste de l'écart-type à partir de l'intervalle inter-quartile

Effets des statistiques robustes

Avant :

```
    [ … blabla … ]
38.999          3.8999
38.996          3.8996
38.996          3.8996
38.997          3.8997
38.995          3.8995
38.998          3.8998
38.996          3.8996
38.997          3.8997
38.997          3.8997
38.995          3.8995
38.998          3.8998
========================
T_tot = 3902.67ms
T_run ~ 39.0267+/-97.2103µs
T_iter ~ 3.90267+/-972.103ns
```

Après :

```
T_tot = 1175.2ms
T_run ~ 38.997+/-0.0022239µs
T_iter ~ 3.8997+/-0.022239ns
```

Le niveau de bruit de mesure est le même (~1-2ns)…

…mais les statistiques robustes permettent d'extraire le « signal utile » !

Conclusion

- Les benchmarks sont un outil utile
- L'écriture d'un benchmark n'a rien d'évident
- Un bon protocole de mesure et d'analyse permet…
  - …d'éliminer un certain nombre de biais
  - …de se rapprocher de la précision de l'horloge machine
- Cette présentation ne fait qu'introduire le problème
  - Gare à vos biais dans l'écriture du code
  - Gare aux biais du matériel (cache, ILP, etc)

Merci de votre attention !

```
Benchmarking addition
T_tot = 279.126ms
T_run ~ 27.865+/-0.000741301µs
T_iter ~ 0.5573+/-0.0104836ns
---------------------
Benchmarking multiplication
T_tot = 278.824ms
T_run ~ 27.865+/-0.000741301µs
T_iter ~ 0.5573+/-0.0104836ns
---------------------
Benchmarking division
T_tot = 195.276ms
T_run ~ 19.507+/-0.000741301µs
T_iter ~ 3.9014+/-0.0104836ns
---------------------
Benchmarking square root
T_tot = 195.223ms
T_run ~ 19.508+/-0.0014826µs
T_iter ~ 3.9016+/-0.0209672ns
```

```
Benchmarking exponential
T_tot = 320.799ms
T_run ~ 32.053+/-0.000741301µs
T_iter ~ 6.4106+/-0.0104836ns
---------------------
Benchmarking logarithm
T_tot = 284.638ms
T_run ~ 28.424+/-0.0252042µs
T_iter ~ 5.6848+/-0.356442ns
---------------------
Benchmarking sinus
T_tot = 524.091ms
T_run ~ 52.365+/-0.0311346µs
T_iter ~ 10.473+/-0.44031ns
---------------------
Benchmarking arc-tangent
T_tot = 549.212ms
T_run ~ 54.879+/-0.014826µs
T_iter ~ 21.9516+/-0.209672ns
```

Mesures effectuées avec un CPU Intel Xeon E5-1620 v3 @ 3.50GHz + Linux 5.5.6 + GCC 9.2.1