Cours X : understanding code performance

Charles Bouillaguet
(charles.bouillaguet@lip6.fr)

2022-XX-XXXX

Famous Quote

Premature optimization
is the root of all evil
— Donald Knuth

Meaning

- Optimization takes (programmer) time
- Use your time efficiently
- Optimization usually makes code more complex
  - Harder to read/debug/maintain/upgrade
- Should be done **only** when necessary

Rule of thumb

- 95% of the running time...
- ... usually concentrated in 5% of code statements
  - E.g. inner loops, specific function, etc.

$\rightsquigarrow$ Need to identify them

Need

- We need tools to **observe** code behavior
- **Where** does it spend most of its time?

Key idea

- **do not** optimize code representing 0.001% of the time
- Common rookie mistake

Where Does my Code Spend Its Time?

Two methods

1. Sampling
2. Instrumentation

Illustration / Case Study

- Colleague from the *Institut de minéralogie, de physique des matériaux et de cosmochimie* (Sorbonne University)
- Least square fitting of model to experimental data points
- **Function** $f : \mathbb{R}^n \to \mathbb{R}^m$
  - f is computed by X million lines of ugly Fortran code
  - $n = 1000$ (parameters to choose)
  - $m = 10^7$ (data points to fit)
- **Goal**: find x that minimizes $\|f(x)\|_2$
- Use `MINPACK` library from 1978 — takes forever. Why?

Sampling

Main Idea

- Just **run** the code
- **Interrupt it** regularly
  - E.g. 1kHz
- Store the **instruction pointer** (“sample”)
- Resume

Afterwards

- Look **where** it was interrupted
- Instruction often executed $\rightsquigarrow$ more samples

```
Samples│92ca:     mov      %ecx,%edx
       │92cc:     sub      0x8(%rsp),%r9
       │92d1:     xor      %eax,%eax
       │92d3:     shr      %edx
       │92d5:     shl      $0x4,%rdx
      2│92d9:     nopl     0x0(%rax)
      3│92e0:     movupd   (%r14,%rax,1),%xmm2
    525│92e6:     movupd   (%r9,%rax,1),%xmm5
    924│92ec:     add      $0x10,%rax
     28│92f0:     mulpd    %xmm5,%xmm2
   2932│92f4:     addsd    %xmm2,%xmm0
      2│92f8:     unpckhpd %xmm2,%xmm2
   3952│92fc:     addsd    %xmm2,%xmm0
       │9300:     cmp      %rax,%rdx
      4│9303:   ↑ jne      92e0 <qrfac_+0x3b0>
       │9305:     mov      0x1c(%rsp),%eax
       │9309:     mov      %ecx,%edx
      2│930b:     and      $0xfffffffe,%edx
       │930e:     add      %edx,%eax
       │9310:     cmp      %edx,%ecx
       │9312:   ↓ je       932e <qrfac_+0x3fe>
       │9314:     lea      0x0(%rbp,%rax,1),%edx
       │9318:     add      %r12d,%eax
       │931b:     movslq   %edx,%rdx
       │931e:     cltq
       │9320:     movsd    (%rbx,%rdx,8),%xmm1
       │9325:     mulsd    (%rbx,%rax,8),%xmm1
      3│932a:     addsd    %xmm1,%xmm0
       │932e:     mov      0x20(%rsp),%rax
      1│9333:     lea      (%rsi,%r13,1),%r9
     17│9337:     divsd    (%rax),%xmm0
       │933b:     lea      0x0(,%r9,8),%rax
```

Case Study

```
Percent │92d3:    shr      %edx
        │92d5:    shl      $0x4,%rdx
   0,02 │92d9:    nopl     0x0(%rax)
   0,03 │92e0:    movupd   (%r14,%rax,1),%xmm2
   4,87 │92e6:    movupd   (%r9,%rax,1),%xmm5
   8,70 │92ec:    add      $0x10,%rax
   0,26 │92f0:    mulpd    %xmm5,%xmm2
  27,60 │92f4:    addsd    %xmm2,%xmm0
   0,02 │92f8:    unpckhpd %xmm2,%xmm2
  37,24 │92fc:    addsd    %xmm2,%xmm0
        │9300:    cmp      %rax,%rdx
   0,04 │9303:    jne      92e0 <qrfac_+0x3b0>
        │9305:    mov      0x1c(%rsp),%eax
```

Pinpointing the Hot Spot

- Loop of 9 CPU instructions
- $\approx 80\%$ of the total running time

Good, but...

- Where is instruction `0x92fc` in my code? hint: `qrfac`

Making Sense of Code Addresses

Where is instruction `0x92fc` in my code?

Good News

- The **debugging symbols** exist to answer this question
- Enable debugging symbols: `gcc [...] -g [...]`
  - Makes executable slightly larger...
- Map between **instructions** and **source file/line**
  - Sometimes not obvious with compiler optimizations
  - May reorder code, permute loops, etc.
- Exploited by many tools: `gdb`, `valgrind`, ...

Simple tool: `addr2line`

```
$ addr2line -e ./speed_lmdif1 0x92fc
minpack/qrfac.c:134 (discriminator 3)
```

Let's look at `minpack/qrfac.c`, line 134

Actually looking at the code

```
void qrfac_(...)
{
    for (int j = 1; j <= n; ++j) {
        // ...
        for (int k = j + 1; k <= n; ++k) {
            double sum = 0;
            for (int i = j; i <= m; ++i)
/* !!! */       sum += a[i + j * m] * a[i + k * m];
            double temp = sum / a[j + j * m];
            for (int i = j; i <= m; ++i)
                a[i + k * m] -= temp * a[i + j * m];
            // ...
        }
        // ...
    }
}
```

Sampling — Summary

(Relative) Ease of Use

- Commercial programs (Intel VTune, ...)
- Under linux: `perf`
  - `$ perf record ./program`
  - `$ perf report`

Advantages

- Runs at $\approx 100\%$ native speed
- Quite precise
- Can measure other things than time

Problems

- Usually requires **administrator** privileges
- And/or cooperation from the operating system

Instrumentation

Main Idea

- **Modify** the code
  - Add measurements
- Run **instrumented** code

Afterwards

- Look at the data collected by the instrumentation

```c
double start = wtime();
double qrfac_time = 0;
int qrfac_calls = 0;
// ...
double qrfac_start = wtime();
qrfac(...);
qrfac_time += wtime() - qrfac_start;
qrfac_calls += 1;
// ...
double total = wtime() - start;
printf("Running time: %.1fs\n", total);
printf("%d calls to qrfac (%.1fs)\n",
    qrfac_calls, qrfac_time);
exit(0);
```

[Manual | Automatic] instrumentation

- Manual: if you know what you're looking for
  - Or if you want your program to print performance results
- Automatic: more heavyweight, simpler

Instrumentation: Goals

What data do we obtain in the end?

- A **Profile**
  - Short summary of performance results
  - Typically per-function
  - Eventually call-graph informations
    - "Function A takes 95% of the time, ..."
    - "...but only when it is called by function B"
- A **Trace**
  - Log of timestamped "events"
  - Visual view of performance problems
  - Very powerful, more complex to use

Can build a profile from a trace, but not the other way around

Instrumentation: the GNU Profiler

Easy and Always Available

Usage

- Compile **and** link with: `gcc [...] -pg [...]`
  - Automatic instrumentation
- Run program

$\rightsquigarrow$ `gmon.out`

- `$ gprof ./program`

GNU Profiler: Result

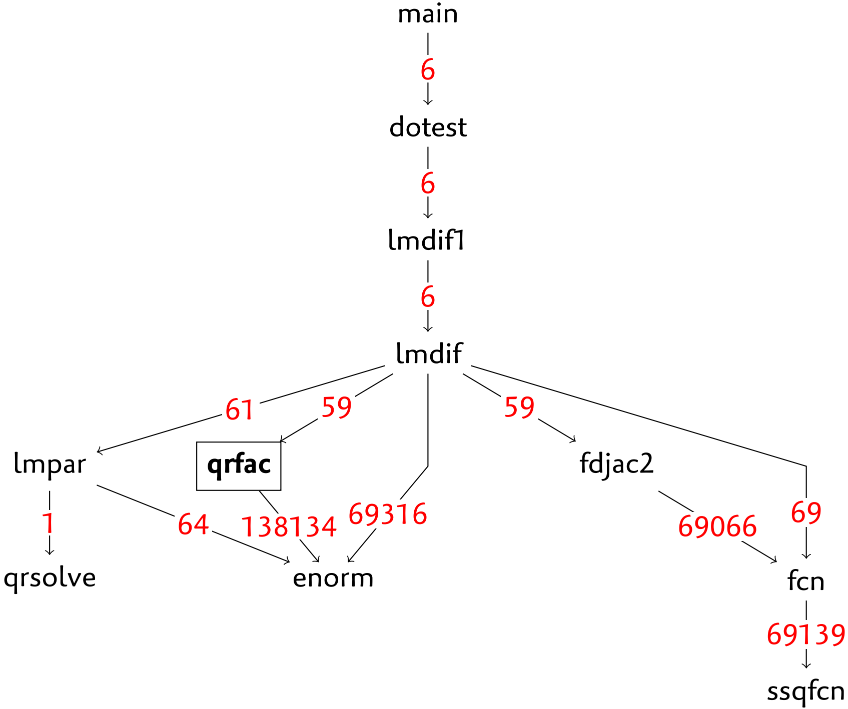
Flat profile

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
97.91	408.82	408.82	59	6.93	6.95	qrfac_
0.91	412.63	3.81	1	3.81	3.81	qrsolv_
0.48	414.64	2.01	69139	0.00	0.00	ssqfcn
0.41	416.35	1.71	207520	0.00	0.00	enorm_
0.17	417.05	0.70	6	0.12	69.59	lmdif_
0.11	417.49	0.44	59	0.01	0.04	fdjac2_
0.02	417.56	0.07	61	0.00	0.06	lmpar_
0.00	417.56	0.00	69133	0.00	0.00	fcn
0.00	417.56	0.00	14	0.00	0.00	wtime
0.00	417.56	0.00	6	0.00	69.59	do_test
0.00	417.56	0.00	6	0.00	0.00	initpt
0.00	417.56	0.00	6	0.00	69.59	lmdif1_

- We learn that 98% of the time is spent in `qrfac`
- Called 59 times, ≈ 6.9s per call

Call Graph

```
index % time    self  children    called     name
...
-----------------------------------------------
                0.70  416.86       6/6           lmdif1_ [3]
[4]    100.0    0.70  416.86       6         lmdif_ [4]
              408.82    1.14      59/59          qrfac_ [5]
                0.07    3.81      61/61          lmpar_ [6]
                0.44    2.01      59/59          fdjac2_ [8]
                0.57    0.00   69316/207520      enorm_ [11]
                0.00    0.00      67/69133       fcn [10]
-----------------------------------------------
              408.82    1.14      59/59          lmdif_ [4]
[5]     98.2  408.82    1.14      59         qrfac_ [5]
                1.14    0.00  138134/207520      enorm_ [11]
-----------------------------------------------
                0.07    3.81      61/61          lmdif_ [4]
[6]      0.9    0.07    3.81      61         lmpar_ [6]
                3.81    0.00       1/1           qrsolv_ [7]
                0.00    0.00      64/207520      enorm_ [11]
-----------------------------------------------
                3.81    0.00       1/1           lmpar_ [6]
[7]      0.9    3.81    0.00       1         qrsolv_ [7]
-----------------------------------------------
                0.44    2.01      59/59          lmdif_ [4]
[8]      0.6    0.44    2.01      59         fdjac2_ [8]
                0.00    2.01   69066/69133       fcn [10]
-----------------------------------------------
...
```


main
6
dotest
6
lmdif1
6
lmdif
61
59
59
lmpar
qrfac
69316
fdjac2
69
1
64
138134
69066
qrsolve
enorm
fcn
69139
ssqfcn

Profiling: Summary

Not just `gprof`

- `score-p`, `scalasca`, `tau`, ...
- Profile more than time (bytes send/s, cache miss, ...)

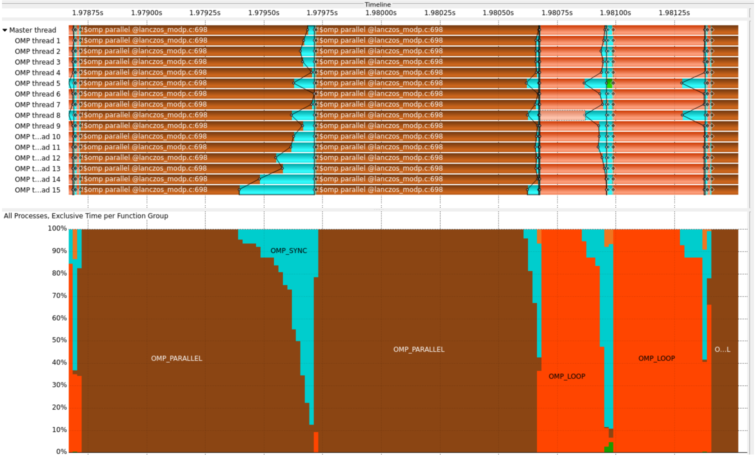
Advantages

- Ease of use (just recompile), **Good first approach**
- No need for `root` privileges
- Call-graph information

Problems

- Coarse grained
  - Does not precisely pinpoint a specific loop
- May slow program down

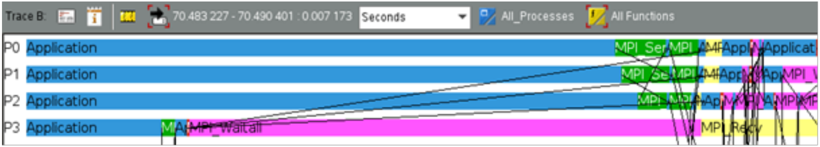
Traces



Multi-thread

- Load imbalance (inactive threads)
- Time spent waiting for sync. (locks, barriers, ...)

Traces



MPI

- Time spent in communications
- What process delays the others?

Traces

Advantages

- Very precise information about dynamic code behavior
- Can be obtained automatically (e.g. `score-p` instrumenter)

Problems

- Visualization tools are mostly commercial (`Vampir`, `itac`)
- Traces can be **large** with many processes/threads