Profiling sequential programs

Key question

- How do I know where my code spends most of its time?
  - This is called "profiling"
  - Many (free and commercial) tools exist
- Baseline tool: GNU gprof
  - Supported by GCC and Intel compilers (and others)
- Linux "perf" infrastructure
- Based on the profile, optimization can be planned
  - Reduction of work
  - Doing work more efficiently

Profiling with gprof

Caveat: all-new profiling tool “gprofng” upcoming!

Profiling with gprof

- Basic sequential profiling tool under Linux: `gprof`
- Compiling for a profiling run

```
$ icx -pg ......
```

- After running the binary, a file `gmon.out` is written to the current directory
- Human-readable output:

```
$ gprof a.out
```

- Inlining should be disabled for profiling
  - But then the executed code isn't what it should be…

Profiling with gprof: Example

- Example with wrapped **double** class:

```
class D {
  double d;
public:
  D(double _d=0) : d(_d) {}
  D operator+(const D& o) {
    D r;
    r.d = d+o.d;
    return r;
  }
  operator double() {
    return d;
  }
};
```

Main program:

```
const int n=10000000;
D a[n],b[n];
D sum;

for(int i=0; i<n; ++i)
  a[i] = b[i] = 1.5;

double s = timestamp();
for(int k=0; k<10; ++k) {
  for(int i=0; i<n; ++i)
    sum = sum + a[i] + b[i];
}
```

Profiling with gprof: Example profiler output

- `icpx -O3 -pg perf.cc`

```
  %    cumulative   self              self     total
 time    seconds   seconds    calls  Ts/call  Ts/call  name
101.01       0.41     0.41                             main
```

- `icpx -O3 -fno-inline -pg perf.cc`

```
  %    cumulative   self              self     total
 time    seconds   seconds    calls  ns/call  ns/call  name
46.44       0.59     0.59 200000000     2.93     4.48  D::operator+(D const&)
29.63       0.96     0.37 240000001     1.56     1.56  D::D(double)
24.82       1.27     0.31                             main
```

- But where did the time *actually* go?
  - Butterfly (callgraph) profile also available
  - Real problem also with use of libraries (STL!)
  - Sometimes you have to roll your own little profiler (see later!)

Flat profile

Called how often?

Time not including callees

Time including callees

```
Each sample counts as 0.000976562 seconds.
  %    cumulative   self              self     total
 time    seconds   seconds    calls   s/call   s/call  name
 66.43       7.96     7.96        1     7.96     9.64  hamilt_
 11.15       9.30     1.34        1     1.34     1.34  outwin_
  8.14      10.28     0.98        1     0.98     1.07  hns_
  2.86      10.62     0.34  5266813     0.00     0.00  zheevx2_
  1.20      10.76     0.14                             __libm_error_support
  1.04      10.89     0.12                             zher2m_
  0.90      11.00     0.11                             cvtas_s_to_a
  0.75      11.09     0.09                             select_
  0.38      11.13     0.05                             cvt_ieee_t_to_text_ex
  0.37      11.18     0.04   445351     0.00     0.00  seclit_
```

Butterfly (call graph) profile

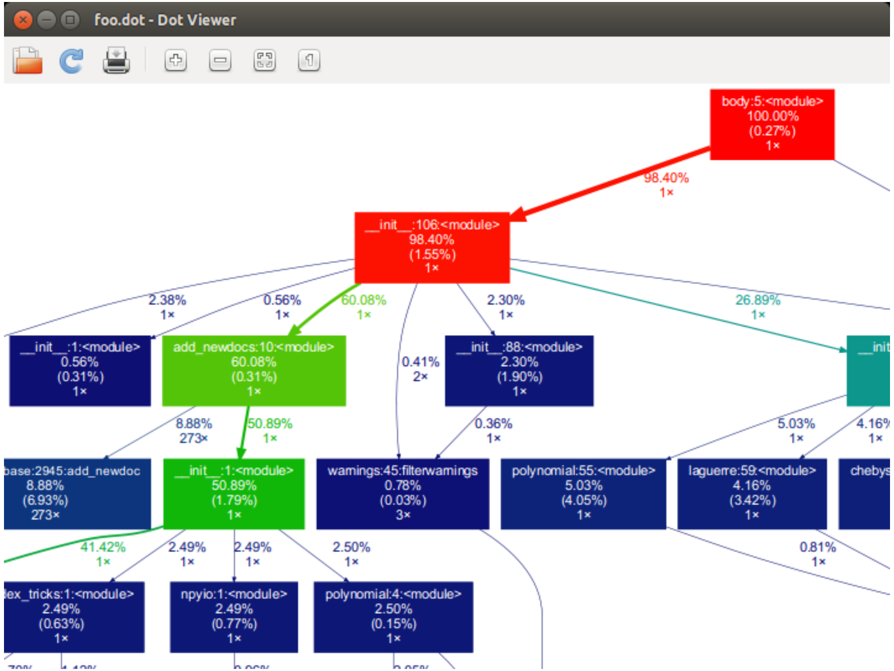
```
granularity: each sample hit covers 2 byte(s) for 20.00% of 0.05 seconds
index % time    self  children    called     name
                                             <spontaneous>
[1]     100.0    0.00    0.05                start [1]
                 0.00    0.05       1/1      main [2]
                 0.00    0.00       1/2      on_exit [28]
                 0.00    0.00       1/1      exit [59]
-----------------------------------------------
                 0.00    0.05       1/1      start [1]
[2]     100.0    0.00    0.05       1        main [2]
                 0.00    0.05       1/1      report [3]
-----------------------------------------------
                 0.00    0.05       1/1      main [2]
[3]     100.0    0.00    0.05       1        report [3]
                 0.00    0.03       8/8      timelocal [6]
                 0.00    0.01       1/1      print [9]
                 0.00    0.01       9/9      fgets [12]
                 0.00    0.00      12/34     strncmp <cycle 1> [40]
                 0.00    0.00       8/8      lookup [20]
                 0.00    0.00       1/1      fopen [21]
                 0.00    0.00       8/8      chewtime [24]
                 0.00    0.00       8/16     skipspace [44]
```

called by

calling

Visualizing the butterfly profile

- Gprof2Dot converts gprof output to graphviz “dot” file
  - https://github.com/jrfonseca/gprof2dot
- View dot file with, e.g., xdot



Profiling MPI programs with gprof

- By design, gprof is a tool for serial code
  - It can, however, be convinced to write a trace file that contains the PID in its name

```
$ GMON_OUT_PREFIX=foo mpirun -np 5 ./a.out
[…]
$ ls
a.out  foo.28219  foo.28220  foo.28221  foo.28222  foo.28223
$ gprof a.out foo.28219
```

  - Accumulating individual files:

```
$ gprof --sum a.out foo.*   # generates gmon.sum
$ gprof a.out gmon.sum
```

  - Take care – all values are summed up across processes!

Sampling-based profiling with Linux perf

Sampling

- Interrupts program in regular intervals
- Records data
  - Where is the program right now (address of “current” instruction)?
    - program function
    - line of code
    - machine instruction
  - What does the call stack look like?
    - How did execution get “here”?
- Results
  - Histogram of “how much time spent where”
  - How much time is spent along a particular call path

Advantages vs. gprof:
- Works on any binary without recompile
- Also captures OS and runtime symbols
- Also works with multi-threaded code

Simple runtime profile with Linux perf

Compile with (Intel compiler; options also work for GCC):

```
$ icx -g -fno-inline -fno-omit-frame-pointer ...
```

Call executable wrapped with perf, generated file **`perf.data`**:

```
$ perf record -g ./a.out
```

Analyze the results with:

```
$ perf report
```

```
Samples: 113K of event 'cycles:Pu', Event count (approx.): 63028635012
  Children      Self  Command  Shared Object         Symbol
+  100,00%     0,00%  a.out    libc.so.6             [.] __libc_start_call_main
+  100,00%     0,00%  a.out    a.out                 [.] main
+   96,01%     0,00%  a.out    a.out                 [.] CG
+   57,51%    57,51%  a.out    a.out                 [.] axpby
+   22,14%    22,14%  a.out    a.out                 [.] dotProduct
+   19,06%    19,06%  a.out    a.out                 [.] applyStencil
+    2,14%     0,00%  a.out    a.out                 [.] computeResidual
+    0,69%     0,64%  a.out    a.out                 [.] __svml_sin4_l9
     0,46%     0,41%  a.out    a.out                 [.] init
     0,23%     0,23%  a.out    a.out                 [.] __intel_avx_rep_memcpy
     0,02%     0,00%  a.out    [unknown]             [.] 0xffffffffb2800c00
```

Output of `perf report`

Inclusive time for function

Self time for function

Event used for profiling

```
Samples: 113K of event 'cycles:Pu', Event count (approx.): 63028635012
  Children      Self  Command  Shared Object        Symbol
+  100,00%     0,00%  a.out    libc.so.6            [.] __libc_start_call_main
+  100,00%     0,00%  a.out    a.out                [.] main
+   96,01%     0,00%  a.out    a.out                [.] CG
+   57,51%    57,51%  a.out    a.out                [.] axpby
+   22,14%    22,14%  a.out    a.out                [.] dotProduct
+   19,06%    19,06%  a.out    a.out                [.] applyStencil
+    2,14%     0,00%  a.out    a.out                [.] computeResidual
+    0,69%     0,64%  a.out    a.out                [.] __svml_sin4_l9
     0,46%     0,41%  a.out    a.out                [.] init
     0,23%     0,23%  a.out    a.out                [.] __intel_avx_rep_memcpy
     0,02%     0,00%  a.out    [unknown]            [.] 0xffffffffb2800c00
```

“profile browser” allows drilling into call tree if “`-g`” option was given for `perf record` (hit “h” for help)

Options for reporting with perf

- Show perf.data in an ncurses browser (TUI) if possible:
  ```
  $ perf report
  ```
- Show perf.data with a column for sample count:
  ```
  $ perf report -n
  ```
- Show perf.data as a text report, with data coalesced and percentages:
  ```
  $ perf report --stdio
  ```
- Disassemble and annotate instructions with percentages (needs some debuginfo):
  ```
  $ perf annotate --stdio
  ```

Some general remarks about profiling

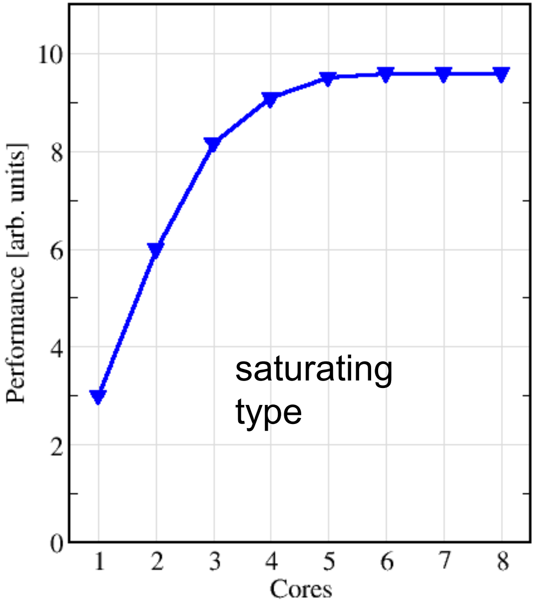
Manual profiling with a timer function

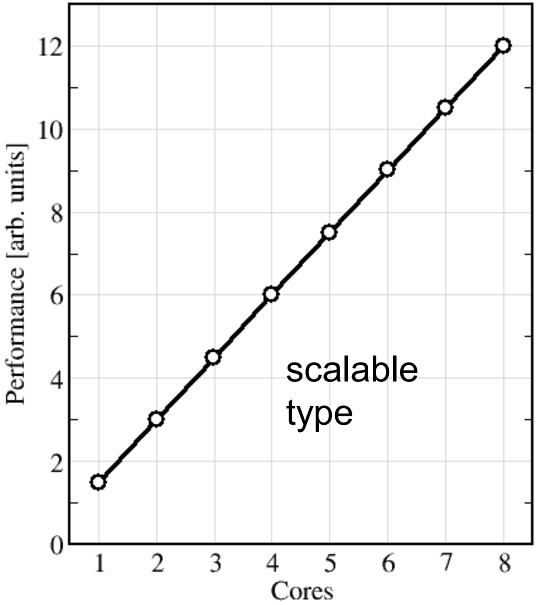
- Measuring walltime on UNIX (-like) systems
  - Stay away from CPU time – it's evil!
  - Use `clock_gettime()` to obtain wall-clock time stamp:

```
#include <time.h>

double getTimeStamp()
{
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return (double)ts.tv_sec + (double)ts.tv_nsec * 1.e-9;
}

double gettimestamp_()
{
    return getTimeStamp();
}
```

Consequences from the saturation pattern for profiling

Clearly distinguish between "saturating" and "scalable" performance on the chip level





Consequences from the saturation pattern for profiling

- Some bottlenecks only show up in parallel execution!
- Code profile for single thread ≠ code profile for multiple threads
  - → Single-threaded profiling may be misleading