HPC.NRW

GPROF TUTORIAL

Application performance analysis with the GNU profiler

October 19, 2020 | Christian Siebert

THE COMPETENCE NETWORK FOR HIGH-PERFORMANCE COMPUTING IN NRW.

INTRODUCTION

Gprof is a free profiler from GNU

- simple way to analyze runtime behaviour of an application
  (low overhead, collect various meaningful insights)
- determine where most of the execution time is spent
  ⇒ locate code regions suited for optimization
- analyzes connections between individual functions
  ⇒ helps in understanding code and suggests elimination of expensive function calls
- part of GNU Binutils and supported by various compilers
  ⇒ available as open-source, almost everywhere
- works for C/C++, Fortran and even Pascal sources

SIMPLE EXAMPLE IN PASCAL (GCD.PAS PAGE 1 OF 2)

```
program gcd; { greatest common divisor }

{... main code calls this function ten million times => omitted ...}
function binary_gcd(a, b: longint) : longint;

   function is_even(x: longint) : boolean;
   begin
      if (x mod 2) = 0 then is_even := true
                       else is_even := false;
   end;

   var
      d : longint;
```

SIMPLE EXAMPLE IN PASCAL (GCD.PAS PAGE 2 OF 2)

```pascal
begin { function binary_gcd(a, b) }
   d := 1;
   while is_even(a) and is_even(b) do
   begin
      a := a div 2; b := b div 2; d := d * 2;
   end;
   while (a <> b) and (a > 0) and (b > 0) do
   begin
      while is_even(a) do a := a div 2;
      while is_even(b) do b := b div 2;
      if a > b then a := (a - b) div 2 else b := (b - a) div 2;
   end;
   binary_gcd := d * a;
end; { function binary_gcd(a, b) }
```

GPROF PREPARATION

step 1) Compile and link source code with option -pg:

```
$ fpc -pg gcd.pas
Free Pascal Compiler version 3.2.0rc1 [2020/02/25] for x86_64
Copyright (c) 1993-2020 by Florian Klaempfl and others
Target OS: Linux for x86-64
Compiling gcd.pas
Linking gcd
69 lines compiled, 0.1 sec
```

step 2) Run instrumented application with some representative input, e.g.:

```
$ ./gcd 14354684 24299194
greatest common divisor of 14354684 and 24299194 is 86474
```

⇒ collects information on runtime behaviour in file gmon.out

GPROF - FLAT PROFILE

step 3a) The Flat Profile shows how much time is spent in each function and how often each function was called.

```
$ gprof --flat-profile gcd
Flat profile (simplified):

Each sample counts as 0.01 seconds.
  %    self             total
time  seconds    calls   s/call  name
79.71  1.10   230000000   0.00  P$GCD$_$BINARY[...]$$_IS_EVEN$LONGINT$$BOOLEAN
19.57  0.27    10000000   0.00  P$GCD_$$_BINARY_GCD$LONGINT$LONGINT$$LONGINT
 0.72  0.01           1   1.38  PASCALMAIN
...
```

$\Rightarrow$ 80% of the total running time is spent in function `is_even()`!

GPROF - CALL GRAPH

step 3b) The Call Graph shows which functions called each other and how many times.

```
$ gprof --graph gcd                                            (simplified)
index % time self   called  name
             0.01       1/1 SYSTEM_$$_SYSENTRY$TENTRYINFORMATION
 [1]   100.0 0.01         1 PASCALMAIN [1]
             0.27  10000000 P$GCD_$$_BINARY_GCD$LONGINT$LONGINT$$LONGINT [2]
-----------------------------------------------
             0.27  10000000 PASCALMAIN [1]
 [2]    99.3 0.27  10000000 P$GCD_$$_BINARY_GCD$LONGINT$LONGINT$$LONGINT [2]
             1.10 230000000 P$GCD$_$[...]$_IS_EVEN$LONGINT$$BOOLEAN [3]
-----------------------------------------------
             1.10 230000000 P$GCD_$$_BINARY_GCD$LONGINT$LONGINT$$LONGINT [2]
 [3]    79.7 1.10 230000000 P$GCD$_$[...]$_IS_EVEN$LONGINT$$BOOLEAN [3]
```

GPROF - ANNOTATED SOURCE

step 3c) Gprof can even annotate your source code. (Add option -g at compile time.)

```
$ gprof --annotated-source gcd
```

```
               function is_even(x: longint) : boolean;
230000000 ->   begin
                 if (x mod 2) = 0 then is_even := true
                                  else is_even := false;
               end;

               var
                 d : longint;

 10000000 ->   begin
                 d := 1;
```

MERGEMAP - REAL-WORLD C++ EXAMPLE (BIOLOGY)

MergeMap constructs consensus genetic maps from a set of individual genetic maps.

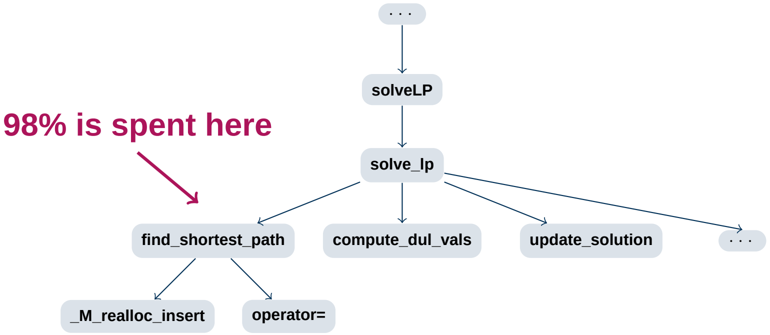
```
$ make CXX="g++" CXXFLAGS="-O2 -g -pg"             (GNU C++ Compiler 8.1.0)
$ MergeMap.exe maps_config
$ gprof --flat-profile MergeMap.exe
```

```
 %     self              total
time  seconds   calls   s/call name
98.44  113.87   1022510  0.00  consensus_map::CG::find_shortest_path(...)
 1.03    1.20 102646561  0.00  void ..._M_realloc_insert<int const&>(...)
 0.29    0.33        21  5.51  consensus_map::CG::solve_lp(double)
 0.13    0.15   1022496  0.00  consensus_map::CG::compute_dual_vals(double)
 0.05    0.06   1022482  0.00  consensus_map::CG::update_solution(...)
 0.03    0.03         7  0.00  consen...::execute_all_pairs_shortest_path()
```

⇒ 98% of the total running time is spent in finding shortest pathes!

MERGEMAP - CALL GRAPH VISUALIZED

```
$ gprof --graph MergeMap.exe                                    (simplified)
```



MCL - REAL-WORLD C EXAMPLE (COMPUTER SCIENCE)

MCL (Markov Cluster Algorithm) is a fast and scalable cluster algorithm for graphs.

```
$ export CC=icc CFLAGS="-O1 -pg"              (Intel C Compiler 19.0.0.117)
$ export LDFLAGS="-pg"
$ ./configure && make
...
$ mcl input.pairs --abc -te 24 -o output.pairs                 (24 Threads)
$ gprof --flat-profile mcl
```

```
 %      self                total
time   seconds    calls    Ks/call  name
42.69  4397.81  114123234    0.00   mclxVectorCompose
25.52  2629.06    2596390    0.00   matrix_vector_array
 7.83   807.04                      __libm_pow_e7
```

⇒ 68% of the total running time is spent in sparse matrix-vector operations!

MUSUBI - REAL-WORLD FORTRAN EXAMPLE (ENGINEERING)

Musubi is the multi-level parallel lattice Boltzmann solver within the APES suite.

```
$ export FC=mpiifort                        (Intel Fortran Compiler 19.0.0.117)
$ export FCFLAGS="-O1 -pg" CFLAGS="-O1 -pg" LINKFLAGS="-O1 -pg"
$ ./waf configure build --target=musubi
$ export MUS_LEVEL=6 MUS_ITER=1000000
$ mpiexec -np 8 ./musubi                                     (8 MPI processes)
$ gprof --flat-profile musubi
```

```
 %      self
time   seconds    calls  name
49.66  3052.05  1000000  mus_auxfieldvar_module_mp_mus_calcauxfield_fluid...
38.40  2359.76  1000000  mus_d3q19_module_mp_bgk_advrel_d3q19_incomp_
 5.71   351.12  1000000  mus_aux_module_mp_mus_update_relaxparams_
```

$\Rightarrow$ 50% of the total running time is spent on computing the auxiliary field!

BONUS: CALL GRAPH VISUALIZATION WITH GPROF2DOT

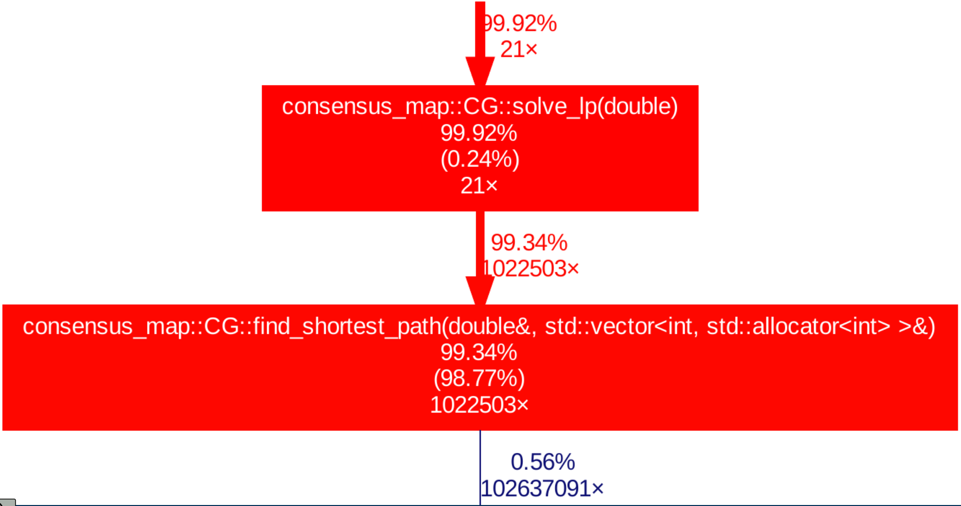
Gprof2dot is a separate tool to visualize call graphs.
It is available on https://github.com/jrfonseca/gprof2dot.

Installation and execution:

```
$ apt-get install python3 python3-pip graphviz
$ pip install gprof2dot
$ MergeMap.exe maps_config
$ gprof MergeMap.exe | gprof2dot | dot -Tpdf -o output.pdf
```

⇒ In conjunction with the dot tool, it can directly generate images (e.g., JPEG or PDF).

BONUS: CALL GRAPH VISUALIZATION WITH GPROF2DOT



99.92%
21×

consensus_map::CG::solve_lp(double)
99.92%
(0.24%)
21×

99.34%
1022503×

consensus_map::CG::find_shortest_path(double&, std::vector<int, std::allocator<int> >&)
99.34%
(98.77%)
1022503×

0.56%
102637091×

SUMMARY

Gprof is a free and easy-to-use profiler

- supported in several programming languages
  e.g., C/C++, Fortran and Pascal ⇒ -pg compile/link option
- works with various compilers
  e.g., demontrated here with GNU and Intel compilers
- low overhead, yet good (first) insights into application behaviour
- intended for sequential application but also works for parallel ones
  e.g., threads and MPI - however, no differentiation of individual threads/processes
- can create flat profile, call graph and source annotations

⇒ good starting tool for performance optimizations