Performance Profiling

What is Performance Profiling?

- The determination of run-time or other performance metrics
  - Net profile: total run-time and metrics (e.g., cache misses)
  - Functional profile: how much time in each function
  - Line profile: how much time on each line of code

- Profiling can be done manually or via a toolkit (or both):
  - Manually measure wall-clock time (e.g., `/usr/bin/time`)
  - Manually add timer functions to code (e.g., `gettimeofday()`)
  - Command-line tools: gprof, cachegrind, HPCToolkit, OpenSpeedshop

When to use PP?

- Profiling is a the first phase of the iterative optimization process:

```
while ( not satisfied with performance ):
    - Acquire performance profile
    - Determine hotspots and other regions to optimize
    - Refactor / optimize code
    - Validate modifications
```

- Focus attention on the ‘hotspots’ (most costly pieces) of the code.
- Why? Cost-benefit is highest.
- Example: function `A()` and `B()` take 75% and 25% of the run-time, respectively.
    - Improving `A()` by 50% yields a net speed-up of 1.6x = 1 / (0.5 * 0.75 + 0.25)
    - Improving `B()` by 50% yields only 1.14x = 1 / (0.75 + 0.5 * 0.25)

Sampling v. Instrumentation

- Most profiling tools use sampling to gauge performance profile.
  - Application is run under control of a (hopefully) lightweight monitoring system and interrupted at a regular interval.
  - The program instruction pointer is recorded each interrupt.
  - After application ends, the recorded program counter information is processed to give an approximation of how long the program spent in certain regions of the code (often by function).
  - Sampling rate of 0.01s is common (e.g., Gprof): too coarse gives lousy statistics; too fine adds too much overhead.
  - This only gives reasonable results if you run the code for many (many) of these sampling intervals!

Example: Gprof

- Ubiquitous profiler from GNU. Not a great tool but easy to use. A good 'first' step in profiling.

1. Compile your code with debugging symbols (`-g`) and for profiling (`-pg`).
2. Run the code for a statistically significant duration (Gprof's rate is 0.01s so you want lots of sample … several minutes is good.)
3. Passed the generated run-time data (`gmon.out`) and your code to Gprof for analysis.

```
[tg459340@login2 hw2]$ gprof --flat ./nbody3
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.29      7.99     7.99                             accel(double*, double*, double*, double*, int)
  0.00      7.99     0.00    50000     0.00     0.00  double const& std::max<double>(double const&, double const&)
  0.00      7.99     0.00    50000     0.00     0.00  double const& std::min<double>(double const&, double const&)
  0.00      7.99     0.00        4     0.00     0.00  double* aligned_alloc<double>(unsigned long)
  0.00      7.99     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z5frandv
  0.00      7.99     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(int, int) [clone
.constprop.0]
…
```

```
[tg459340@login2 hw2]$ gprof --graph ./nbody3
                     Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 0.13% of 7.99 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    7.99    0.00                 accel(double*, double*, double*, double*, int) [1]
-----------------------------------------------
                0.00    0.00   50000/50000       search(double*, double*, double*, double*, int) [23]
[14]     0.0    0.00    0.00   50000         double const& std::max<double>(double const&, double const&) [14]
-----------------------------------------------
                0.00    0.00   50000/50000       search(double*, double*, double*, double*, int) [23]
[15]     0.0    0.00    0.00   50000         double const& std::min<double>(double const&, double const&) [15]
-----------------------------------------------
                0.00    0.00       4/4           double* Allocate<double>(double*&, unsigned long) [25]
[16]     0.0    0.00    0.00       4         double* aligned_alloc<double>(unsigned long) [16]
-----------------------------------------------
                0.00    0.00       1/1           __libc_csu_init [33]
[17]     0.0    0.00    0.00       1         _GLOBAL__sub_I__Z5frandv [17]
-----------------------------------------------
                0.00    0.00       1/1           __libc_csu_init [33]
[18]     0.0    0.00    0.00       1         __static_initialization_and_destruction_0(int, int) [clone .constprop.0] [18]
-----------------------------------------------
…
```

```
[tg459340@login2 hw2]$ gprof --line ./nbody3
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 73.56     24.02    24.02                             accel(double*, double*, double*, double*, int) (nbody3.cpp:72 @ 4017f0)
 10.78     27.55     3.52                             accel(double*, double*, double*, double*, int) (nbody3.cpp:74 @ 401806)
  2.52     28.37     0.82                             accel(double*, double*, double*, double*, int) (nbody3.cpp:74 @ 401812)
  2.21     29.09     0.72                             accel(double*, double*, double*, double*, int) (nbody3.cpp:71 @ 4017d7)
  2.06     29.76     0.67                             accel(double*, double*, double*, double*, int) (nbody3.cpp:75 @ 401816)
  2.03     30.42     0.66                             accel(double*, double*, double*, double*, int) (nbody3.cpp:75 @ 40180a)
  1.74     30.99     0.57                             accel(double*, double*, double*, double*, int) (nbody3.cpp:68 @ 4017bd)
  1.47     31.47     0.48                             accel(double*, double*, double*, double*, int) (nbody3.cpp:70 @ 4017d2)
  1.47     31.95     0.48                             accel(double*, double*, double*, double*, int) (nbody3.cpp:76 @ 40181a)
  1.41     32.42     0.46                             accel(double*, double*, double*, double*, int) (nbody3.cpp:76 @ 40180e)
  0.61     32.62     0.20                             accel(double*, double*, double*, double*, int) (nbody3.cpp:72 @ 4017cd)
  0.26     32.70     0.09                             accel(double*, double*, double*, double*, int) (nbody3.cpp:69 @ 4017b6)
  0.12     32.74     0.04                             accel(double*, double*, double*, double*, int) (nbody3.cpp:68 @ 4017b0)
  0.06     32.76     0.02                             accel(double*, double*, double*, double*, int) (nbody3.cpp:69 @ 4017c8)
```

Sampling v. Instrumentation

- Instrumentation is the process of inserting timers or other inquiry functions in the code. Can record *events* or *hardware counters*.
- This can be manually done, fully automated, or guided.
  - Many tools (e.g., CrayPat) automatically instrument your object code.
  - Some advanced tools (e.g., Intel *Vtune Advisor*) allow you to select code regions for instrumentation.
- Run the application with instrumentation and then analyze the resulting metrics.
  - Often much more than functional or line run-time profiles.
  - A single counter is often not useful. But several counters together can give good insight on cache efficiency, floating-point usage, etc.
- Warning! Instrumented code can take much (much!) longer to run.

Hardware Counters

- Hardware counters available on most modern hardware.
- Some are directly counted; some are derived from multiple others.
- Gives a detailed view of very low-level operations at a hardware level.
  - Vector and Scalar floating-point operations (fadd, fdiv, sqrt)
  - Total clock cycles
  - L1 cache hits/misses
- Only can record a few counters at a time so often must repeat measurements multiple times to get all desired details.
- PAPI (http://icl.cs.utk.edu/papi) is a common interface to hardware counters.
  - PAPI API can be used to manually instrument but often used by other profiling tools (e.g., OpenSpeedShop, HPCToolkit, PerfExpert)
  - Analyzing the counters is expert-level stuff. High-level toolkits are useful for humanizing the data.

```
[tg459340@login2 ~]$ papi_avail
Available events and hardware information.
--------------------------------------------------------------------------------
PAPI Version             : 5.3.0.0
Vendor string and code   : GenuineIntel (1)
Model string and code    : Intel(R) Xeon(R) CPU E5-2680 0 @ 2.70GHz (45)
CPU Revision             : 7.000000
CPUID Info               : Family: 6  Model: 45  Stepping: 7
CPU Max Megahertz        : 2699
CPU Min Megahertz        : 2699
Hdw Threads per core     : 1
Cores per Socket         : 8
Sockets                  : 2
NUMA Nodes               : 2
CPUs per Node            : 8
Total CPUs               : 16
Running in a VM          : no
Number Hardware Counters : 11
Max Multiplex Counters   : 32
--------------------------------------------------------------------------------
Name        Code    Avail Deriv Description (Note)
PAPI_L1_DCM  0x80000000  Yes   No   Level 1 data cache misses
PAPI_L1_ICM  0x80000001  Yes   No   Level 1 instruction cache misses
PAPI_L2_DCM  0x80000002  Yes   Yes  Level 2 data cache misses
PAPI_L2_ICM  0x80000003  Yes   No   Level 2 instruction cache misses
PAPI_L3_DCM  0x80000004  No    No   Level 3 data cache misses
PAPI_L3_ICM  0x80000005  No    No   Level 3 instruction cache misses
PAPI_L1_TCM  0x80000006  Yes   Yes  Level 1 cache misses
PAPI_L2_TCM  0x80000007  Yes   No   Level 2 cache misses
PAPI_L3_TCM  0x80000008  Yes   No   Level 3 cache misses
…
PAPI_FML_INS 0x80000061  No    No   Floating point multiply instructions
PAPI_FAD_INS 0x80000062  No    No   Floating point add instructions
PAPI_FDV_INS 0x80000063  Yes   No   Floating point divide instructions
PAPI_FSQ_INS 0x80000064  No    No   Floating point square root instructions
PAPI_FNV_INS 0x80000065  No    No   Floating point inverse instructions
PAPI_FP_OPS  0x80000066  Yes   Yes  Floating point operations
PAPI_SP_OPS  0x80000067  Yes   Yes  Floating point operations; optimized to count scaled single precision vector operations
PAPI_DP_OPS  0x80000068  Yes   Yes  Floating point operations; optimized to count scaled double precision vector operations
PAPI_VEC_SP  0x80000069  Yes   Yes  Single precision vector/SIMD instructions
PAPI_VEC_DP  0x8000006a  Yes   Yes  Double precision vector/SIMD instructions
PAPI_REF_CYC 0x8000006b  Yes   No   Reference clock cycles
-------------------------------------------------------------------------
Of 108 possible events, 50 are available, of which 17 are derived.
```

Example: PerfExpert

- Created by the folks at TACC!
- Runs your application several times to collect specific hardware counters.
- Provides high-level analysis of metrics.
- Provides optimization recommends (but they are rather generic).
- Can even try to optimize your source code for you.
  - Careful … it’ll squash your source code!

```
[tg459340@c557-404 hw2]$ perfexpert 0.1 ./nbody3 " -n 2000 -s 200"
[perfexpert] Collecting measurements [hpctoolkit]
[perfexpert]    [1] 6.493576278 seconds (includes measurement overhead)
[perfexpert]    [2] 6.437754005 seconds (includes measurement overhead)
[perfexpert]    [3] 6.462963367 seconds (includes measurement overhead)
[perfexpert] Analysing measurements
--------------------------------------------------------------------------------
…
Loop in function accel(double*, double*, double*, double*, int) in nbody3.cpp:65 (99.83% of the total runtime)
================================================================================
ratio to total instrns    %  0.........25.........50.........75.........100
 - floating point      100.0 **************************************************
 - data accesses        24.1 ************
* GFLOPS (% max)        24.7 ************
 - packed               12.2 ******
 - scalar               12.5 ******
--------------------------------------------------------------------------------
performance assessment  LCPI good......okay.......fair.......poor.......bad
* overall               2.95 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
* data accesses         1.56 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
 - L1d hits             0.84 >>>>>>>>>>>>>>>>>
 - L2d hits             0.72 >>>>>>>>>>>>>>
 - L3d hits             0.00
 - LLC misses           0.00
* instruction accesses  0.00
 - L1i hits             0.00
 - L2i hits             0.00
 - L2i misses           0.00
* data TLB              0.00
* instruction TLB       0.00
* branch instructions   0.03 >
 - correctly predicted  0.03 >
 - mispredicted         0.00
* floating-point instr  8.02 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
 - slow FP instr        2.76 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
 - fast FP instr        5.26 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
```

Other Toolkits:

- Intel Advisor (Vectorization, memory usage)
- Intel Amplifier (Threading)
- Intel Trace Analyzer and MPI Snapshot (MPS)
  - Tracing provides a time history of the call graph. Very useful for parallel code optimization since you can see when some ranks are waiting on others.
- HPCToolkit and OpenSpeedShop are good open-source sampling and counter (via PAPI) tools.
- Cachegrind / Callgrind (part of valgrind).
  - Visualize results with Kcachegrind (KDE) GUI.

```
[tg459340@c568-013 mpi]$ mps -f stat_20161130-171359
| Parsing stat_20161130-171359/stat-2.bin file.
| …
| Parsing stat_20161130-171359/stat-6.bin file.
| Done.
|
| Function summary for all ranks
|-----------------------------------------------------------------------------------
|            Function        Time(sec)           Time(%)       Volume(MB)       Volume(%)              Calls
|-----------------------------------------------------------------------------------
           MPI_Bcast             8.56             49.01          3906.25          100.00              32768
      MPI_Comm_split             4.94             28.28             0.00            0.00               8208
            MPI_Init             3.12             17.84             0.00            0.00                 16
       MPI_Comm_free             0.82              4.72             0.00            0.00               8208
       MPI_Allreduce             0.02              0.14             0.02            0.00                128
       MPI_Comm_rank             0.00              0.01             0.00            0.00               4128
       MPI_Comm_size             0.00              0.00             0.00            0.00                 33
|===================================================================================
| TOTAL                         17.47            100.00          3906.27          100.00              53489
```

```
[tg459340@c568-013 mpi]$ mps stat_20161130-171359
| Summary information
|--------------------------------------------------------------------
  Application      : ./mpi_matmul
  Number of ranks  : 16
  Used statistics  : stat_20161130-171359
  Creation date    : 2016-11-30 17:14:00
|
  Your application is not well optimized.
  OpenMP weak parallelism.
|
  WallClock time :             5.21 sec
| Total application lifetime. The time is elapsed time for the slowest process.
|
      MPI Time:                   1.09 sec           21.01%
|     Time spent inside the MPI library. High values are usually bad.
|     This value is AVERAGE. The application is Communication-bound.
          MPI Imbalance:                0.09 sec             1.74%
|         This value is LOW. The application workload is well balanced between
|
      Computation Time:           4.11 sec           78.99%
|     This value is AVERAGE. The application is Computation-bound.
|         OpenMP Time:                  0.00 sec             0.00%
|         This value is NEGLIGIBLE.
          Serial Time:                  4.11 sec            78.99%
|         This value is HIGH. This application is NOT well parallelized via
| Disk Usage for all processes
         Data read:       786.4  KB
      Data written:         2.8  MB
     I/O wait time:         0.00 sec ( 0.00 %)
|
  Peak memory consumption :           43.80 MB (rank 11)
  Mean memory consumption :           42.98 MB
```