Performance Analysis: Single-Node

The importance of profiling

- The **Golden Rules** of profiling:
  - **Profile your code**
    - The compiler/runtime will NOT do all the optimisation for you.
  - **Profile your code yourself**
    - Don't believe what anyone tells you. They're wrong.
  - **Profile on the hardware you want to run on**
    - Don't profile on your laptop if you plan to run on HECToR.
  - **Profile your code running the full-sized problem**
    - The profile will almost certainly be qualitatively different for a test case.
  - **Keep profiling your code as you optimise**
    - Concentrate your efforts on the thing that slows your code down.
    - This will change as you optimise.
    - So keep on profiling.

What we talk about here

- **You've already seen how to profile codes**
- **Here we will show you some extra features**
  - These are most useful for improving single node performance, e.g.
    - a serial code, running on just one core
    - a single rank of a parallel (prob. MPI) job
      - that rank is running on one core of a node
    - an OpenMP code running threads on the cores of a single node
    - one rank of a hybrid MPI/OpenMP code
      - that rank is running threads on the cores of a single node
  - So we are not looking at profiling usage of the network or I/O here.
- **What we are looking at:**
  - loop-level profiling
  - hardware performance counters
  - hybrid MPI/OpenMP jobs
  - memory footprint and overheads

Loop statistics

- **Function-level tracing**
  - Identifies routines taking most time; candidates for optimisation
  - But, where in the routine is time spent?
    - on one single task or spread across many small tasks?
- **Loop-level tracing gives finer granularity**
  - Most HPC codes are based round loopnests
    - If one loopnest dominates runtime, can think about rewriting it
    - If lots of loopnests contribute, maybe look at combining them
- **Information provided**
  - Inclusive loop times
    - How much time was spent executing that loopnest
  - Loop tripcounts (min, mean, max)
    - Can be used to give compiler hints to allow additional optimisation
    - tripcounts very interesting if considering a GPU port
- **Only available with Cray compiler**
  - Compiler adds some additional features into binary executable.
  - Should be done as separate experiment
    - compiler optimizations are restricted with this feature
    - this will affect runtime and probably also skew the profile

Collecting loop statistics

- **Get the right modules**
  - `PrgEnv-cray` because this is a CCE-specific feature
  - `perftools` because we need profiling
- **Re-compile & link code**
  - Add `-h profile_generate` flag to compile and link lines
- **Instrument binary executable for tracing using one of:**
  - `pat_build -u my_program`
    - traces all user functions; fine for small codes
  - `pat_build -w -T func1_,func2_... my_program`
    - traces specified functions, better for larger programs
      - watch out for trailing underscores (pat_build will say how many functions traced)
  - `pat_build` using an existing `apa` file.
- **Run the instrumented application**
  - `aprun <options> my_program+pat`
- **Create report with loop statistics**
  - `pat_report my_program+pat+<etc>.xf > loops_report`

Default report

Table 2: Loop Stats by Function (from -hprofile_generate)

Loop Incl Time Total	Loop Hit	Loop Trips Avg	Loop Trips Min	Loop Trips Max	Function=/.LOOP[.] PE=HIDE
6.067347	100	25	0	25	sweepy_.LOOP.1.li.32
6.067182	2500	25	0	25	sweepy_.LOOP.2.li.33
5.877503	50	25	0	25	sweepz_.LOOP.05.li.48
5.877408	1250	25	0	25	sweepz_.LOOP.06.li.49
2.971128	50	25	0	25	sweepx2_.LOOP.1.li.28
2.971051	1250	25	0	25	sweepx2_.LOOP.2.li.29
2.947443	50	25	0	25	sweepx1_.LOOP.1.li.28
2.947376	1250	25	0	25	sweepx1_.LOOP.2.li.29
2.247618	187500	107	0	107	riemann_.LOOP.2.li.63

Perfectly nested loops:
- Incl time almost same
- Outer Hit*Avg = Inner Hit
- Consecutive source lines

Hardware performance counters

- **CrayPAT can interface with Cray XC30's HWPCs**
  - Gives extra information on how hardware is behaving
  - Very useful for understanding (& optimising) application performance
- **Provides information on**
  - hardware features, e.g. caches, vectorisation and memory bandwidth
- **Available on per-program and per-function basis**
  - Per-function information only available through tracing
- **Number of simultaneous counters limited by hardware**
  - 4 counters available with Intel Ivybridge processors
  - If you need more, you'll need multiple runs
- **Most counters accessed through the PAPI interface**
  - Either native counters or derived metrics constructed from these

Hardware counters selection

- **HWPCs collected using CrayPAT**
  - Compile and instrument code for profiling as before

- **Set `PAT_RT_HWPC` environment variable at runtime**
  - e.g. in the your job script
    - Hardware counter events are not collected by default (except with APA)
- **`export PAT_RT_HWPC=...`**
  - either a list of named PAPI counters
  - or `<set number>` = a pre-defined (and useful) set of counters
    - recommended way to use HWPCs
    - there are around 20 groups to choose from
      - To see them:
        - `pat_help -> counters -> ivybridge -> groups`
        - `man hwpc`
        - `more ${CRAYPAT_ROOT}/share/CounterGroups.intel_fam6mod62`

Technical term for Ivybridge

- **HW counter information shown by `pat_report`:**
  - Raw data
  - Derived metrics
  - Desirable thresholds

Predefined Ivybridge HW Counter Groups

Default is number 1 with CrayPAT APA procedure

0: D1 with instruction counts
1: Summary -- cache and TLB metrics
2: D1, D2, L3 Metrics
6: Micro-op queue stalls
7: Back end stalls
8: Instructions and branches
9: Instruction cache
10: Cache Hierarchy

~~11: Floating point operations (unsupported)~~
~~12: AVX floating point operations (unsupported)~~
~~13: SSE and AVX floating point operations SP (unsupported)~~
~~14: SSE and AVX floating point operations DP (unsupported)~~
19: Prefetchs
23: Cache metrics (same as 1)

Example: HW counter data and derived metrics

```
 USER / evolve_
------------------------------------------------------------------------
  Time%                                                  1.6%
  Time                                               0.909054 secs
  Imb. Time                                          0.057555 secs
  Imb. Time%                                             6.4%
  Calls                                0.116M/sec     187500.0 calls
  PAPI_L1_DCM                         18.108M/sec     29376518 misses
  PAPI_TLB_DM                          0.007M/sec        11643 misses
  PAPI_L1_DCA                        170.243M/sec    276182686 refs
  PAPI_FP_OPS                                                0 ops
  DATA_CACHE_REFILLS_FROM_L2_OR_NORTHBRIDGE:
    ALL                               18.711M/sec     30354680 fills
  DATA_CACHE_REFILLS_FROM_NORTHBRIDGE  0.003M/sec         5084 fills
  User time (approx)                   1.622 secs   3731260602 cycles  100.0% Time
  HW FP Ops / User time                                      0 ops    0.0%peak(DP)
  HW FP Ops / WCT
  Computational intensity               0.00 ops/cycle    0.00 ops/ref
  MFLOPS (aggregate)                    0.00M/sec
  TLB utilization                   23720.03 refs/miss  46.328 avg uses
  D1 cache hit,miss ratios             89.4% hits        10.6% misses
  D1 cache hit,refill ratio            89.0% hits        11.0% refills
  D1 cache utilization (misses)         9.40 refs/miss   1.175 avg hits
  D1 cache utilization (refills)        9.10 refs/refill 1.137 avg uses
  D2 cache hit,miss ratio             100.0% hits         0.0% misses
  D1+D2 cache hit,miss ratio          100.0% hits         0.0% misses
….
```

Some hints on interpreting the data

- **TLB utilization**
  - Memory loaded in pages: 4kB standard (could use larger hugepages)
  - e.g. 512 x 8-byte double precision floats
  - So if every double was used once, expect 512 refs/miss
    - Less than 512 shows poor use; more than 512 is good (**5420.38** excellent)
      - N.B. <avg uses> = <refs/miss> / 512

- **D1 cache utilization**
  - Level 1 cache line is 64 contiguous bytes, e.g. 8 x 8-byte doubles
  - So if every double was used once, expect 8 refs/miss
    - Corresponds to hit ratio of 87.5% [i.e. 100*(1 - 1/<refs/miss>)]
      - N.B. <avg uses> = <refs/miss> / 8
    - Less than 8 (or 87.5%) shows poor use
    - Rule of thumb: want this to be 20 (or 95%) or more

- **D1+D2 cache hit ratio**
  - Should be high (rule of thumb is more than 97%);

CrayPAT observations and suggestions

D1 + D2 cache utilization: 39.8% of total execution time was spent in 4 functions with combined D1 and D2 cache hit ratios below the desirable minimum of 97.0%. Cache utilization might be improved by modifying the alignment or stride of references to data arrays in these functions.

D1_D2_cache_hit_ratio	Time%	Function
56.8%	12.0%	calc3_
77.9%	6.4%	calc2_
95.7%	1.4%	calc1_
96.3%	20.0%	calc3_.LOOP@li.80

TLB utilization: 19.6% of total execution time was spent in 3 functions with fewer than the desirable minimum of 512 data references per TLB miss. TLB utilization might be improved by modifying the alignment or stride of references to data arrays in these functions.

LS_per_TLB_DM	Time%	Function
2.56	12.0%	calc3_
5.32	6.3%	calc2_

Interpreting the performance numbers

- **Performance numbers are an average over all ranks**
  - explains non-integer values
- **This does not always make sense**
  - e.g. if ranks are not all doing the same thing:
    - Master-slave schemes
    - MPMD apruns combining multiple, different programs
- **Want them to only process data for certain ranks**
  - `pat_report -sfilter_input='condition' ...`
  - `condition` should be an expression involving `pe`, e.g.
    - `pe<1024` for the first 1024 ranks only
    - `pe%2==0` for every second rank

  - This option is also useful for large `.ap2` or `.xf` data files
    - Compiling a report may take too long
    - or exceed the available memory

OpenMP data collection and reporting

- **Give finer-grained profiling of threaded routines**
  - Measure overhead incurred entering and leaving
    - Parallel regions
      - #pragma omp parallel
    - Work-sharing constructs within parallel regions
      - #pragma omp for

- **Timings and other data now shown per-thread**
  - rather than per-rank

- **OpenMP tracing enabled with `pat_build -gomp ...`**
  - CCE, PGI: insert tracing points around parallel regions automatically
  - Intel, Gnu, Pathscale: need to use CrayPAT API manually

- **Can also use sampling without API**
  - Get a per-rank view; no per-thread counters
  - Run with `OMP_NUM_THREADS=1` during sampling

OpenMP data collection and reporting

- **Load imbalance for hybrid MPI/OpenMP programs**
  - now calculated across all threads in all ranks
  - imbalances for MPI and OpenMP combined
    - Can choose to see imbalance in each programming model separately
    - See next slide for details
- **Data displayed by default in pat_report**
  - no additional options needed
  - Report focuses on where program is spending its time
  - Assumes all requested resources should be used
    - you may have reasons not to want to do this, of course

Imbalance options for data display (`pat_report -O …`)

- **These options control how load balance is displayed:**

- **`profile_pe_th` (default view)**
  - Imbalance based on the set of all threads in the program
  - i.e. imbalance from OpenMP and MPI combined
    - this is best measure to understand code performance

- **`profile_pe.th`**
  - Highlights imbalance across MPI ranks
  - Thread data for each rank is aggregated
    - max used rather than mean, to highlight under-performers
  - Aggregated thread data merged into MPI rank data

- **`profile_th_pe`**
  - For each thread, show imbalance over MPI ranks
  - Example: Load imbalance shows where thread 4 in each MPI rank didn’t get much work

Memory usage

- **Knowing how much memory each rank uses is important:**
  - What is the minimum number of cores I can run this problem on?
    - given there is 32GB (~30GB usable) of memory per node (32 cores)
  - Does memory usage scale well in the application?
  - Is memory usage balanced across the ranks in the application?
  - Is my application spending too much time allocating and freeing?
- **Profile heap usage using CrayPAT**
  - `pat_build -gheap ...`

Heap statistics

Memory per rank
~30GB usable memory per node

Too many allocs/frees?
Would show up as ETC time in CrayPAT report

Memory leaks
Not usually a problem in HPC

```
Notes for table 5:

  Table option:
    -O heap_hiwater
  Options implied by table option:
    -d am@,ub,ta,ua,tf,nf,ac,ab -b pe=[mmm]

  This table shows only lines with Tracked Heap HiWater MBytes > 0.


Table 5:  Heap Stats during Main Program

 Tracked |   Total |  Total | Tracked | Tracked |PE[mmm]
    Heap |  Allocs |  Frees | Objects |  MBytes |
 HiWater |         |        |     Not |     Not |
  MBytes |         |        |   Freed |   Freed |

   9.794 |     915 |    910 |       4 |   1.011 |Total
|-----------------------------------------------------
|   9.943 |    1170 |   1103 |      68 |   1.046 |pe.0
|   9.909 |     715 |    712 |       3 |   1.010 |pe.22
|   9.446 |    1278 |   1275 |       3 |   1.010 |pe.43
|=====================================================
```

Summary

- **Profiling is essential to identify performance bottlenecks**
  - even at single core level
- **CrayPAT has some very useful extra features**
  - can pinpoint and characterise the hotspot loops (not just routines)
  - hardware performance counters give extra insight into performance
  - well-integrated view of hybrid programming models
    - most commonly MPI/OpenMP
    - also CAF, UPC, SHMEM, pthreads, OpenACC, CUDA
  - information on memory usage
- **And remember the Golden Rules**
  - including the one about not believing what anyone tells you

Are there any questions?

```
pat_record(PAT_STATE_OFF);
```