ARCHER Single Node Optimisation

Profiling

Slides contributed by Cray and EPCC

What is profiling?

- Analysing your code to find out the proportion of execution time spent in different routines.

- Essential to know this if we are going to target optimisation.

- No point optimising routines that don’t significantly contribute to the overall execution time.
  - can just make your code less readable/maintainable

Code profiling

- Code profiling is the first step for anyone interested in performance optimisation
- Profiling works by instrumenting code at compile time
  - Thus it’s (usually) controlled by compiler flags
  - Can reduce performance
- Standard profiles return data on:
  - Number of function calls
  - Amount of time spent in sections of code
- Also tools that will return hardware specific data
  - Cache misses, TLB misses, cache re-use, flop rate, etc…
  - Useful for in-depth performance optimisation

Sampling and tracing

- Many profilers work by sampling the program counter at regular intervals (normally 100 times per second).
  - low overhead, little effect on execution time
- Builds a statistical picture of which routines the code is spending time in.
  - if the run time is too small (< ~10 seconds) there aren't enough samples for good statistics
- Tracing can get more detailed information by recording some data (e.g. time stamp) at entry/exit to functions
  - higher overhead, more effect on runtime
  - unrestrained use can result in huge output files

Standard Unix profilers

- Standard Unix profilers are prof and gprof
- Many other profiling tools use same formats
- Usual compiler flags are `-p` and `-pg`:
  - `ftn -p mycode.F90 -o myprog` for prof
  - `cc -pg mycode.c -o myprog` for gprof
- When code is run it produces instrumentation log
  - `mon.out` for prof
  - `gmon.out` for gprof
- Then run prof/gprof *on your executable program*
  - eg. `gprof myprog` (*not* `gprof gmon.out`)

Standard profilers

- **prof myprog** reads **mon.out** and produces this:

```
%Time Seconds Cumsecs  #Calls   msec/call  Name
 32.4    0.71    0.71      14     50.7     relax_
 28.3    0.62    1.33      14     44.3     resid_
 11.4    0.25    1.58       3     83.      __f90_close
  5.9    0.13    1.71 1629419      0.0001  _mcount
  5.0    0.11    1.82  339044      0.0003  __f90_slr_i4
  5.0    0.11    1.93  167045      0.0007
 __inrange_single
  2.7    0.06    1.99     507      0.12    _read
  2.7    0.06    2.05       1     60.      MAIN_
```

Standard profilers

- **`gprof myprog`** reads **`gmon.out`** and produces something very similar
- **`gprof`** also produces a program calltree sorted by inclusive times
- Both profilers list all routines, including obscure system ones
  - Of note: **`mcount`**(), _**`mcount`**(), **`moncontrol`**(), _**`moncontrol`**() **`monitor`**() and _**`monitor`**() are all overheads of the profiling implementation itself
  - _**`mcount`**() is called every time your code calls a function; if it's high in the profile, it can indicate high function-call overhead
  - **`gprof`** assumes calls to a routine from different parents take the same amount of time – may not be true

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 ARCHER.

- **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.

CrayPAT

- Can do both statistic sampling and function/loop level tracing.

Recommended usage:

1. Build and instrument code
2. Run code and get statistic profile
3. Re-instrument based on profile
4. Re-run code to get more detailed tracing

Example with CrayPAT (1/2)

- Load performance tools software

```
module load perftools
```

- Re-build application (keep .o files)

```
make clean
make
```

- Instrument application for automatic profiling analysis
  - You should get an instrumented program `a.out+pat`

```
pat_build -O apa a.out
```

- Run the instrumented application (...+pat) to get top time consuming routines
  - You should get a performance file ("`<sdatafile>.xf`") or multiple files in a directory `<sdatadir>`

Example with CrayPAT (2/2)

- Generate text report and an `.apa` instrumentation file

  ```
  pat_report [<sdatafile>.xf | <sdatadir>]
  ```

  - Inspect the `.apa` file and sampling report whether additional instrumentation is needed
    - See especially sites “Libraries to trace” and “HWPC group to collect”

- Instrument application for further analysis (a.out+apa)

  ```
  pat_build –O <apafile>.apa
  ```

- Run application (...+apa)
- Generate text report and visualization file (.ap2)

  ```
  pat_report –o my_text_report.txt <data>
  ```

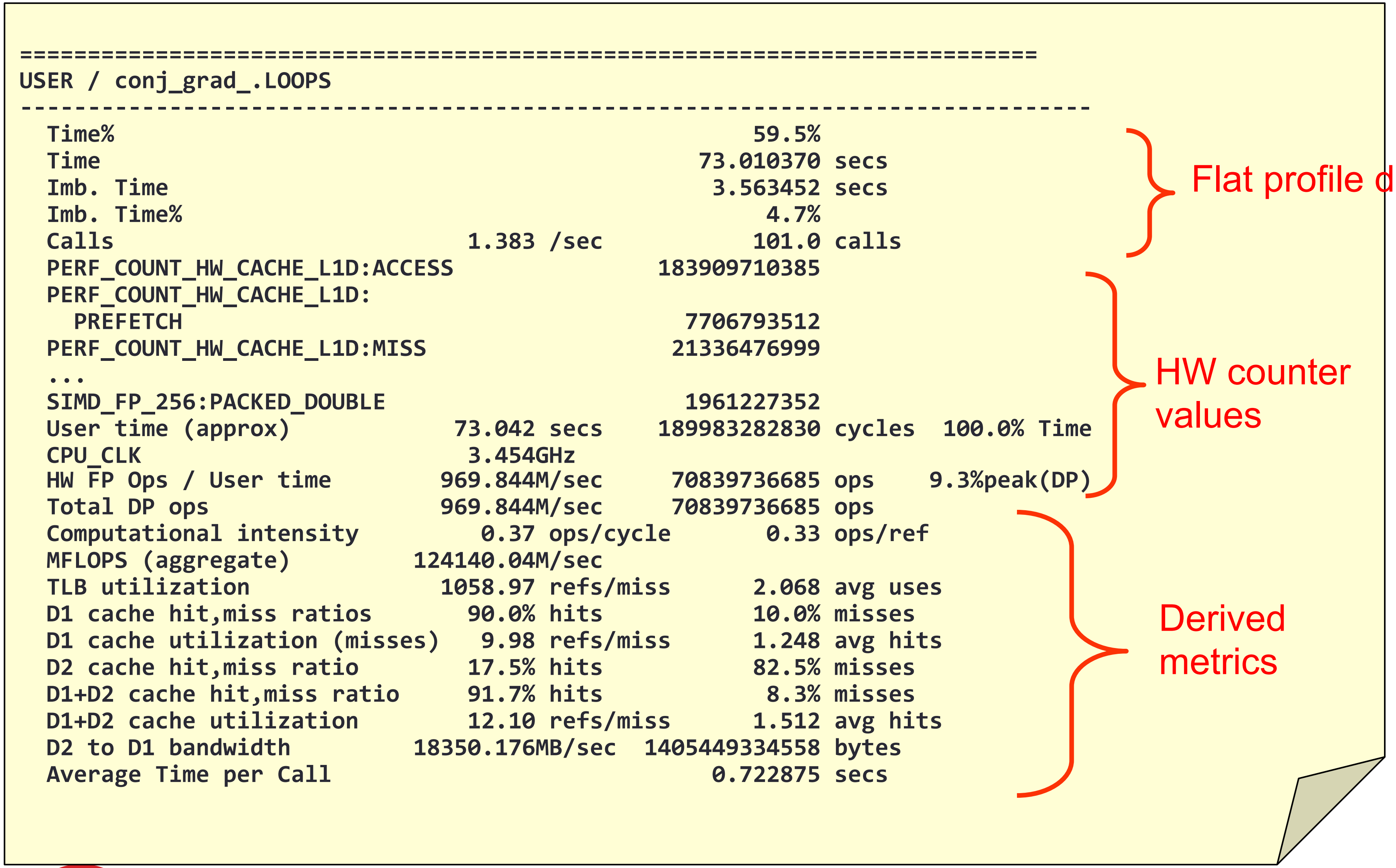
- View report in text and/or with Cray Apprentice^2

  ```
  app2 <datafile>.ap2
  ```

Finding single-core hotspots

- Remember: pay attention only to user routines that consume significant portion of the total time
- View the key hardware counters, for example
  - L1 and L2 cache metrics
  - use of vector (SSE/AVX) instructions
  - Computational intensity (= ratio of floating point ops / memory accesses)

- CrayPAT has mechanisms for finding “the” hotspot in a routine (e.g. in case the routine contains several and/or long loops)
  - CrayPAT API
    - Possibility to give labels to “PAT regions”
  - Loop statistics (works only with Cray compiler)
    - Compile & link with CCE using -h profile_generate
    - pat_report will generate loop statistics if the flag is enabled



```
===========================================================================
USER / conj_grad_.LOOPS
-------------------------------------------------------------------------------
  Time%                                                  59.5%
  Time                                               73.010370 secs
  Imb. Time                                           3.563452 secs
  Imb. Time%                                              4.7%
  Calls                          1.383 /sec              101.0 calls
  PERF_COUNT_HW_CACHE_L1D:ACCESS                  183909710385
  PERF_COUNT_HW_CACHE_L1D:
    PREFETCH                                        7706793512
  PERF_COUNT_HW_CACHE_L1D:MISS                     21336476999
  ...
  SIMD_FP_256:PACKED_DOUBLE                         1961227352
  User time (approx)            73.042 secs    189983282830 cycles  100.0% Time
  CPU_CLK                        3.454GHz
  HW FP Ops / User time       969.844M/sec      70839736685 ops    9.3%peak(DP)
  Total DP ops                969.844M/sec      70839736685 ops
  Computational intensity         0.37 ops/cycle       0.33 ops/ref
  MFLOPS (aggregate)        124140.04M/sec
  TLB utilization             1058.97 refs/miss       2.068 avg uses
  D1 cache hit,miss ratios       90.0% hits           10.0% misses
  D1 cache utilization (misses)   9.98 refs/miss      1.248 avg hits
  D2 cache hit,miss ratio        17.5% hits           82.5% misses
  D1+D2 cache hit,miss ratio     91.7% hits            8.3% misses
  D1+D2 cache utilization        12.10 refs/miss      1.512 avg hits
  D2 to D1 bandwidth        18350.176MB/sec  1405449334558 bytes
  Average Time per Call                           0.722875 secs
```

Flat profile data

HW counter values

Derived metrics

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 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

Predefined Ivybridge HW Counter Groups

Default is number 1 with CrayPAT APA procedure

0: D1 with instruction counts

1: Summary -- FP and cache 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 dispatched

12: AVX floating point operations

13: SSE and AVX floating point operations SP

14: SSE and AVX floating point operations DP

19: Prefetchs

23: FP and cache metrics (same as 1)

```
 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

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: insert tracing points around parallel regions automatically
  - Intel, Gnu: need to use CrayPAT API manually

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

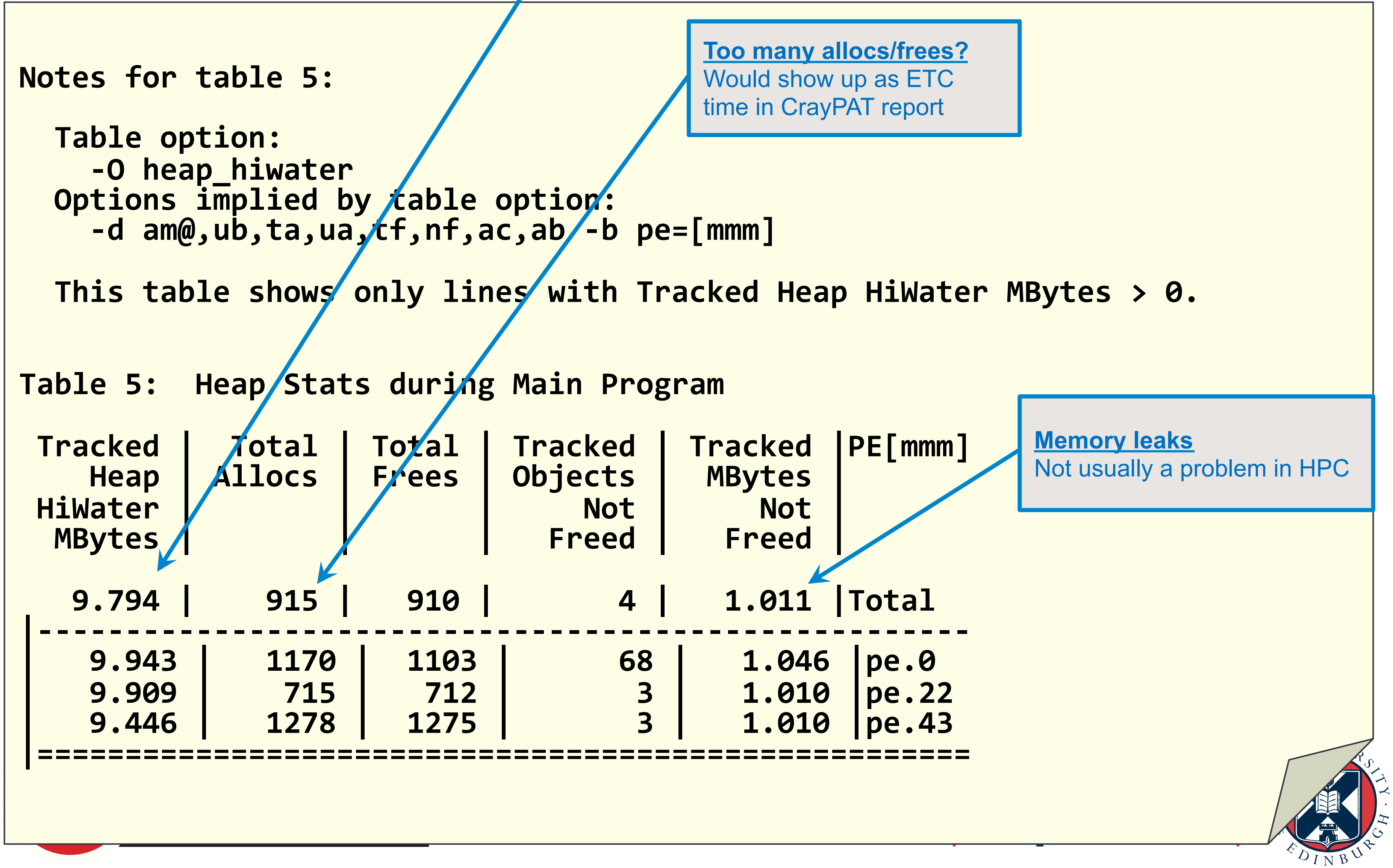
- `profile_th_pe`
  - For each thread, show imbalance over MPI ranks
  - Example: Load imbalance shows where thread 4 in each MPI rank didn't get much work

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



```
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
|=====================================================
```

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

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