



NATURAL
ENVIRONMENT
RESEARCH COUNCIL

EPSRC

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 ($< \sim 10$ 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)`



Analysis and Profiling



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	<code>relax_</code>
28.3	0.62	1.33	14	44.3	<code>resid_</code>
11.4	0.25	1.58	3	83.	<code>__f90_close</code>
5.9	0.13	1.71	1629419	0.0001	<code>_mcount</code>
5.0	0.11	1.82	339044	0.0003	<code>__f90_slr_i4</code>
5.0	0.11	1.93	167045	0.0007	
					<code>__inrange_single</code>
2.7	0.06	1.99	507	0.12	<code>_read</code>
2.7	0.06	2.05	1	60.	<code>MAIN_</code>



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-base
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 <apafilename>.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 Apprentice²
`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



- 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 / remap_

```
-----  
Time%                25.2%  
Time                15.801180 secs  
Imb. Time          2.582609 secs  
Imb. Time%         14.7%  
Calls              0.026M/sec      460,800.0 calls  
CPU_CLK_UNHALTED:THREAD_P  77,964,376,624  
CPU_CLK_UNHALTED:REF_P    2,689,572,161  
DTLB_LOAD_MISSES:MISS_CAUSES_A_WALK  20,626,569  
DTLB_STORE_MISSES:MISS_CAUSES_A_WALK  17,745,058  
L1D:REPLACEMENT          2,753,483,367  
L2_RQSTS:ALL_DEMAND_DATA_RD  1,912,839,218  
L2_RQSTS:DEMAND_DATA_RD_HIT  1,757,495,428  
FP_COMP_OPS_EXE:SSE_SCALAR_DOUBLE      1,597  
FP_COMP_OPS_EXE:SSE_FP_SCALAR_SINGLE  1,556,036,610  
FP_COMP_OPS_EXE:X87          1,878,388,524  
FP_COMP_OPS_EXE:SSE_PACKED_SINGLE     302,976,589  
SIMD_FP_256:PACKED_SINGLE    5,003,127,724  
User time (approx)      17.476 secs      47,202,147,918 cycles 100.0% Time  
CPU_CLK              2.90GHz  
HW FP Ops / User time  2,556.183M/sec   44,671,354,883 ops  11.8% peak(DP)  
Total SP ops          2,448.698M/sec   42,792,964,761 ops  
Total DP ops          107.485M/sec     1,878,390,122 ops  
MFLOPS (aggregate)    61,348.39M/sec  
D2 cache hit,miss ratio  94.4% hits      5.6% misses  
D2 to D1 bandwidth    6,680.690MiB/sec 122,421,709,963 bytes  
Average Time per Call  0.000034 secs  
CrayPat Overhead : Time 11.4%
```

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_PERFCTR` environment variable at runtime
 - e.g. in the job script
 - Hardware counter events are **not** collected by default (except with APA)
- `export PAT_RT_PERFCTR=...`
 - 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 15 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)



Example: Group 2

USER / sweepy_

Time%		14.6%	
Time		8.738150	secs
Imb. Time		3.077320	secs
Imb. Time%		27.2%	
Calls	11.547 /sec	100.0	calls
CPU_CLK_UNHALTED:THREAD_P		92,754,888,918	
CPU_CLK_UNHALTED:REF_P		2,759,876,135	
L1D:REPLACEMENT		1,813,741,166	
L2_RQSTS:ALL_DEMAND_DATA_RD		1,891,459,700	
L2_RQSTS:DEMAND_DATA_RD_HIT		1,644,133,800	
LLC_MISSES		98,952,928	
LLC_REFERENCES		690,626,471	
User time (approx)	8.660 secs	23,390,899,520	cycles 100.0% Time
CPU_CLK	3.36GHz		
D2 cache hit,miss ratio	86.4% hits	13.6%	misses
L3 cache hit,miss ratio	85.7% hits	14.3%	misses
D2 to D1 bandwidth	13,330.757MiB/sec	121,053,420,792	bytes
Average Time per Call		0.087381	secs
CrayPat Overhead : Time	0.0% ...		

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



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?



Heap statistics

Memory per rank
~30GB usable memory per node

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

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

