HPCToolkit Workshop: Introduction to Application Performance Analysis on Linux Systems

or

What I learned at a recent workshop

Johan Larsson
Center for Turbulence Research
Why profile?

- Peak performance of modern systems is rarely achieved in practice
  - Peak rate = (cycles / second) * (flops / cycle)
  - Linpack benchmark (top 500 list) typically at 70-80% of peak flop-rate
  - Science codes generally much lower:
    - Only floating-point operations count, integer operations don’t
    - Must have ordered structure (e.g., multiply, then add, for 2 flops/cycle)
    - Memory latency and bandwidth generally limiting
  - More complex architectures => bigger gap to peak performance?
- Some examples of performance:
  - *Miranda*, 65K cores on BG/L, incompressible R-T (Poisson eqn, FFT)
    - 2.76 Tflops, roughly 1% of peak rate
  - *Hybrid*, 3K cores on Cray XT-4, shock/turbulence interaction (fully explicit)
    - Estimated 1.5 Tflops, roughly 10-12% of peak rate
  - Note: different architectures and algorithms => not fair comparison
Why profile?

- Most of us have no idea of our code’s performance -- profiling tells you, and points out where improvements might be made
  - Sometimes obvious, sometimes less so…
- Learn a bit about the implications of programming decisions

Why use HPCToolkit?
- Disclaimer: I haven’t tried other profiling tools
- No changing (instrumentation) of the source code
- Analyze the results on my laptop afterwards
- GUI shows source code along with profiling information

Why not use HPCToolkit?
- University product: support and bug-fixing?
- Not available everywhere
  - Expected on Cray XT (CNL) later this year
  - BG/P?
HPCToolkit: how does it work?

- Uses statistical sampling:
  - No change to source code
  - During execution, interrupt 100-1000 times per second and log status of execution at that time
  - Relies on PAPI hardware performance counters

- Analyze executable:
  - Compiler may have changed code -- try to un-unroll loops etc to correlate with source code

- Analysis of final database:
  - GUI that shows performance metrics and correlates with source code

- What's needed on the cluster?
  - PAPI
  - HPCToolkit (except the viewer)
HPCToolkit: overview

- [http://www.hipersoft.rice.edu/hpctoolkit/index.html](http://www.hipersoft.rice.edu/hpctoolkit/index.html) (main page)
- [http://hipersoft.rice.edu/hpctoolkit/SERVER/sc04/index.html](http://hipersoft.rice.edu/hpctoolkit/SERVER/sc04/index.html) (tutorial)
- 4 steps on cluster: compile; run; analyze executable; build database
- 1 step on laptop/desktop: analyze database
HPCToolkit: compile code

- **Module load papi/3.5.0 (on Jacquard)**
- Compile code as per usual, except:
  - Use `-g1` (or similar) for source-code line information
  - Don’t use `-ipa` (or similar) for inter-procedural optimization
  - Note that `-Ofast` may turn on `-ipa` -- instead manually set all optimizations
• Can log up to 4 PAPI counters each execution
• Find available counters by (on Jacquard) `papi_avail`:

The following correspond to fields in the PAPI_event_info_t structure.

<table>
<thead>
<tr>
<th>Name</th>
<th>Code</th>
<th>Avail</th>
<th>Deriv</th>
<th>Description (Note)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI_L1_DCM</td>
<td>0x80000000</td>
<td>Yes</td>
<td>Yes</td>
<td>Level 1 data cache misses</td>
</tr>
<tr>
<td>PAPI_L1_ICM</td>
<td>0x80000001</td>
<td>Yes</td>
<td>Yes</td>
<td>Level 1 instruction cache misses</td>
</tr>
<tr>
<td>PAPI_L2_DCM</td>
<td>0x80000002</td>
<td>Yes</td>
<td>No</td>
<td>Level 2 data cache misses</td>
</tr>
<tr>
<td>PAPI_L2_ICM</td>
<td>0x80000003</td>
<td>Yes</td>
<td>No</td>
<td>Level 2 instruction cache misses</td>
</tr>
<tr>
<td>PAPI_L3_DCM</td>
<td>0x80000004</td>
<td>No</td>
<td>No</td>
<td>Level 3 data cache misses</td>
</tr>
<tr>
<td>PAPI_L3_ICM</td>
<td>0x80000005</td>
<td>No</td>
<td>No</td>
<td>Level 3 instruction cache misses</td>
</tr>
<tr>
<td>PAPI_L1_TCM</td>
<td>0x80000006</td>
<td>Yes</td>
<td>Yes</td>
<td>Level 1 cache misses</td>
</tr>
<tr>
<td>PAPI_L2_TCM</td>
<td>0x80000007</td>
<td>Yes</td>
<td>Yes</td>
<td>Level 2 cache misses</td>
</tr>
<tr>
<td>PAPI_L3_TCM</td>
<td>0x80000008</td>
<td>No</td>
<td>No</td>
<td>Level 3 cache misses</td>
</tr>
<tr>
<td>PAPI_CA_SNP</td>
<td>0x80000009</td>
<td>No</td>
<td>No</td>
<td>Requests for a snoop</td>
</tr>
<tr>
<td>PAPI_CA_SHR</td>
<td>0x8000000a</td>
<td>No</td>
<td>No</td>
<td>Requests for exclusive access to shared cache line</td>
</tr>
<tr>
<td>PAPI_CA_CLN</td>
<td>0x8000000b</td>
<td>No</td>
<td>No</td>
<td>Requests for exclusive access to clean cache line</td>
</tr>
<tr>
<td>PAPI_CA_INV</td>
<td>0x8000000c</td>
<td>No</td>
<td>No</td>
<td>Requests for cache line invalidation</td>
</tr>
<tr>
<td>PAPI_CA_ITV</td>
<td>0x8000000d</td>
<td>No</td>
<td>No</td>
<td>Requests for cache line intervention</td>
</tr>
<tr>
<td>PAPI_L3_LDM</td>
<td>0x8000000e</td>
<td>No</td>
<td>No</td>
<td>Level 3 load misses</td>
</tr>
<tr>
<td>PAPI_L3_STM</td>
<td>0x8000000f</td>
<td>No</td>
<td>No</td>
<td>Level 3 store misses</td>
</tr>
<tr>
<td>PAPI_BRU_IDL</td>
<td>0x80000010</td>
<td>No</td>
<td>No</td>
<td>Cycles branch units are idle</td>
</tr>
<tr>
<td>PAPI_FXU_IDL</td>
<td>0x80000011</td>
<td>No</td>
<td>No</td>
<td>Cycles integer units are idle</td>
</tr>
<tr>
<td>PAPI_FPU_IDL</td>
<td>0x80000012</td>
<td>Yes</td>
<td>No</td>
<td>Cycles floating point units are idle</td>
</tr>
<tr>
<td>PAPI_LSU_IDL</td>
<td>0x80000013</td>
<td>No</td>
<td>No</td>
<td>Cycles load/store units are idle</td>
</tr>
</tbody>
</table>

…
HPCToolkit: run code

- Modify your batch-script:

```
mptexec -n 2 hpcrun -e PAPI_TOT_CYC:1000000 -e PAPI_L2_DCM:100000
-e PAPI_FP_OPS:500000 -e PAPI_TLB_DM:100000 ./hybrid.exe
```

- Here I’m sampling:
  - Total cycles, every 1000000 cycles
  - L2 cache misses, every 100000 cycles
  - Floating point operations, every 500000 cycles
  - Translation lookaside buffer misses, every 100000 cycles
- My code ran for 50-100 seconds -- for longer runs, sample less often
- This produces a few trace-files per process, like:

```
-rw------- 1 jola jola  2649 Jul 25 22:11 hybrid.exe-10a013c-1804-2.csp
-rw------- 1 jola jola  2649 Jul 25 22:11 hybrid.exe-10a013c-1804-1.csp
-rw------- 1 jola jola  2649 Jul 25 22:11 hybrid.exe-10a013c-1795-2.csp
-rw------- 1 jola jola  2649 Jul 25 22:11 hybrid.exe-10a013c-1795-1.csp
-rw------- 1 jola jola 1945569 Jul 25 22:11 hybrid.exe-10a013c-1795-0.csp
-rw------- 1 jola jola  1564 Jul 25 22:11 hybrid.exe-10a013c-1795.csl
-rw------- 1 jola jola 1960605 Jul 25 22:11 hybrid.exe-10a013c-1804-0.csp
-rw------- 1 jola jola  1511 Jul 25 22:11 hybrid.exe-10a013c-1804.csl
```
HPCToolkit: analyze executable / build database

- On front-end node, analyze executable by simply running:
  
  hpcstruct ./hybrid.exe > ./hybrid.exe.struct

- On front-end node, build database by:

  hpcprof -S ./hybrid.exe.struct -l $HOME/hybrid_source_code/ hybrid.exe-10a013c-1804-0.csp

- Notes:
  - Use largest trace-files -- the small ones are ? (system-related?)
  - One database per process -- in reality, pick some relevant one(s)
HPCToolkit: explore database

- Move generated database to laptop
- Download *HPCViewer* from http://outreach.scidac.gov
- Open database in *HPCViewer* and explore sampled metrics and generate derived metrics
- Some useful derived metrics:
  - Percent of peak: flops / (2 * cycles) (replace 2 by proper value)
  - L2 misses / flops -- 1 in 100 OK, higher not as good
  - …
Example: *Hybrid* code on the *Jacquard* Opteron cluster

- The *Hybrid* code:
  - Finite-differences, explicit in time/space, C++, MPI with asynchronous communication (initiate send/receive; compute interior; finalize communication; compute near-boundary points)
  - Potential for load-imbalance due to solution-adaptive algorithm
  - Excellent weak scaling due to explicitness and hidden communication