Profiling Tutorial

Soon Yung Jun (Fermilab, SCD/Physics and Detector Simulation Group)
LArSoft Tools and Technology Workshop
20 June 2017, Fermilab
Debugging is finally done! Ready for a test drive?
My program runs, but seems very slow …

CPU
Throughput
Instruction stall

Memory
Latency
Cache misses

I/O
Communication
DB contention

Multithreading
Load balancing
Scalability

Where to start?
Typical Software Development Cycle

- Design Algorithm
- Coding
- Tuning
- Debugging
- Testing Validation
- Profiling Optimization
This Tutorial

• PART-I
  – A brief introduction to computing performance profiling
  – An overview of selected profiling tools and examples
• PART-II
  – Profiling results of the LArTest application with IgProf and OpenSShop
• Demos and Questions
PART-I

Introduction to Computing Performance Profiling
Overview of Selected Profiling Tools and Examples
Why Profile?
Performance tuning is an essential part of the development cycle

• Free lunch is over as modern hardware architectures are getting more complex and parallel
• HEP applications are usually complicated too
• Every $/Watt matters (computing with a limited budget)
• Understanding the code performance is responsibility of the software developer
• Maximize CPU flop rate and minimize memory operations (balancing them is not an easy task)
Computing Performance Profiling and Analysis

• Performance benchmarking quantifies usage/changes of CPU time and memory (amount required or churn)

• Performance profiling analyzes
  – Hot spots, bottlenecks and efficient utilization of resources
  – Code efficiency (instruction/cycle, latencies, I/O and etc.)

• Identifying opportunities for optimization
Understanding Computer Performance

- Hardware platform (processors) popularly used in HEP
  - CISC (x86), RISC (ARM), MIC, GPU(SIMT), FPGA
- Speed: cycle vs. frequency
  - cycle time = 1/(clock frequency)
  - 2.0 GHz = 0.5 ns per cycle
  - CPU Time = \( \sum \)(number of clock cycles)/frequency
- Memory: latency vs. bandwidth
  - latency: the time interval between the request for information and the access (to the first bit of that information)
  - Bandwidth: the number of bits per second
- Throughput vs. locality: CPI, MIPS, FLOPS, FMO
- Pipelining: instruction throughput, data dependency, ILP, ...
Understanding CPU Performance

• Q1: Which operation takes more cycles?
  1. Integer division
  2. Double division
  3. Function call
  4. static_cast<int>(double)

• Strategies
  – Do not mix data type
  – Avoid unnecessary divisions and function calls in the inner most loop
Understanding Memory Transaction

- Memory hierarchy
  - Registers
  - Cache (L1/L2)
  - DRAM (rss)
  - Virtual (vsize)
  - Secondary storage

- Caching
  - Spatial locality (data storage, coalescence)
  - Temporal locality (data reusability in near future)
  - Replacement polices
  - TLB (translation look-aside buffer, the most recent page access)
Understanding Memory Transaction

• Example of memory accesses scenarios

• Do not over-optimize by yourself, but rely on profiling first
Understanding Memory Performance

Q2: Which ratio is the biggest in memory access?
1. L1 Cache/Register
2. L2 Cache/L1 Cache
3. RAM/L2 Cache
4. Virtual Memory/RAM

Strategies
- Try to fit everything in RAM
- Try to fit essential calculations in cache
Basic Concepts of Performance Profilers

• Program segments:
  – Code
  – Stack (program)
  – Heap

• Collecting program events
  – Hardware interrupts
  – Code instrumentation
  – Instruction set simulation
  – Tracing (when)

• Periodic sampling
  – Top of the stack (exclusive)
  – Anywhere in the stack (inclusive)
Classification of Profilers by Techniques used

- **Instrumentation**: inserts extra code at each function call to count how many times the function is called and how much time it takes.

- **Sampling**: The profiler tells the operating system to generate an interrupt and counts how many times an interrupt occurs in each part of the program
  - no modification of the program
  - time-based
  - event-based

- **Debugging tools**: The profiler inserts temporary debug breakpoints at every function or every code line (valgrind)
Examples of Profilers

• Basic OS tools:
  – gprop/perf
  – cachegrind/callgrind

• Hardware counter
  • PAPI and tools set

• Vendor tools
  – Intel VTune Amplifier XE, Inspector, Advisor, ITAC
  – AMD CodeAnalyst
  – Allinea (map and DDD)

• ASCR tools (Open source)
  – HPCToolkit (Rice Univ.)
  – TAU (Oregon Univ.)
  – OpenISpeedshop (Krell)

• HEP
  – FAST (FNAL)
  – IgProf
  – Gooda
gprof: demo

- Compile your program with gcc using –pg flag
- Run your program (as usual) – will produce gmon.out
- Run gprof

```bash
wget https://g4cpt.fnal.gov/g4p/demos/demo.cc

g++ -pg demo.cc -o demo

time ./demo

gprof ./demo
```

Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th>time</th>
<th>cumulative</th>
<th>self</th>
<th>self</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>seconds</td>
<td>seconds</td>
<td>calls</td>
<td>ms/call</td>
<td>ms/call</td>
</tr>
<tr>
<td>36.49</td>
<td>0.51</td>
<td>0.51</td>
<td>1</td>
<td>514.51</td>
</tr>
<tr>
<td>34.34</td>
<td>1.00</td>
<td>0.48</td>
<td>1</td>
<td>484.25</td>
</tr>
<tr>
<td>30.05</td>
<td>1.42</td>
<td>0.42</td>
<td>1</td>
<td>423.72</td>
</tr>
<tr>
<td>0.00</td>
<td>1.42</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>1.42</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
</tr>
</tbody>
</table>
PAPI (Performance API)

- A standard API to access hardware performance counters
- Relation between software performance and processor events
- Event metrics
  - FLOPS, Load/Store
  - cache hit/miss, TLB miss
  - power consumption (MuMMI)
  - platform specific metrics
Hardware Counters

- Operating systems support both non-derived and derived PAPI presets: `papi_avail-a` for listing
- A list of some possible hardware counter combinations

<table>
<thead>
<tr>
<th>For Xeon processors:</th>
<th>For Opteron processors:</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI_FP_INS, PAPI_LD_INS, PAPI_SR_INS</td>
<td>Load store info, memory bandwidth needs</td>
</tr>
<tr>
<td>PAPI_L1_DCM, PAPI_L1_TCA</td>
<td>L1 cache hit/miss ratios</td>
</tr>
<tr>
<td>PAPI_L2_DCM, PAPI_L2_TCA</td>
<td>L2 cache hit/miss ratios</td>
</tr>
<tr>
<td>LAST_LEVEL_CACHE_MISSES, LAST_LEVEL_CACHE_REFERENCES</td>
<td>L3 cache info</td>
</tr>
<tr>
<td>MEM_UNCORE_RETIRED:REMOTE_DRAM, MEM_UNCORE_RETIRED:LOCAL_DRAM</td>
<td>Local/nonlocal memory access</td>
</tr>
<tr>
<td>PAPI_FAD_INS, PAPI_FML_INS</td>
<td>Floating point add multiply</td>
</tr>
<tr>
<td>PAPI_FDV_INS, PAPI_FSQ_INS</td>
<td>Square root and divisions</td>
</tr>
<tr>
<td>PAPI_FP_OPS, PAPI_VEC_INS</td>
<td>Floating point and vector instructions</td>
</tr>
<tr>
<td>READ_REQUEST_TO_L3_CACHE:ALL_CORES, L3_CACHE_MISSES:ALL_CORES</td>
<td>L3 cache</td>
</tr>
</tbody>
</table>
Example of Sampling Tools and Workflow: HPCToolkit

- Typically unmodified binary and call stack analysis
- Code centric view, GUI and text-based flat profile
Example of Integrated Tools: TAU (Tuning Analysis Utilities)

- Dynamic, compiler based, source based Instrumentation
- Analysis tools
  - ParaProf
  - PerfExplorer
  - Tracer (Jumpshot, vampir)
- Various built-in graphical tools
- Dis/advantage: compiler/source-based instrumentation
Understanding Sampling Profilers

• Q3: What are the disadvantages of sampling profilers?
  1. Sampling uncertainty (→ statistical analysis with repetition)
  2. Non-Reproducibility (→ use definitive tools)
  3. Interference from other processes (→ standalone nodes)
  4. Jumping between cores (→ setting NUMA affinity, pinning)
  5. All of above

• Strategies
  – Understand your program first (intensity: arithmetic vs. memory)
  – Overview with sampling experiments
  – Focus on critical parts of code: Rule of 80:20
  – Detailed optimization with hardware counter experiments
  – Benchmarking and monitoring of every minor/major update
PART-II

Profiling Results of LArTest with IgProf and OpenISpeedshop
Application for this Tutorial: LArTest

- A standalone Geant4 application (developed by H. Wenzel)
  - Cubic (5mx5mx5m) LAr fiducial volume
  - GDML to assign step limits and sensitive detector to volumes
  - Optical (scintillation) photons produced in sensitive detector
- Computing performance monitoring features
  - Event time
  - Memory (IgProf, statm)
  - Statistics of the number for tracks/steps per particle type
- Profiling examples with IgProf and OpenI|Speedshop
Installing and Running IgProf

- Installation: [http://igprof.org/install.html](http://igprof.org/install.html)
- Implementation

```c
if (void *sym = dlsym(0, "igprof_dump_now")) {
    dump_ = __extension__ (void(*)(const char *)) sym;
} else { /* message */ ; }
```

- Running igprof on your application (-mp: memory profiling) [http://igprof.org/running.html](http://igprof.org/running.html)

```bash
igprof -d -mp -z -o ${IG_OUT} $exe {args...}
```

- Analysis (web-navigable version of the report, -r for $mode)

```
cmd="igprof-analyse --sqlite -d -v -g -r"
$cmd ${mode} ${IG_OUT} | sqlite3 out.sql3
```

- ${mode} = MEM_LIVE, MEM_MAX, MEM_TOTAL
IgProf

- Snapshot live memory on the heap (for every N-events)
  ```c
  if ( dump_ && evt->GetEventID() % 25 == 0) {
    sprintf(outfile, "|gzip -9c > IgProf.%d.gz", evt->GetEventID()+1);
    dump_(outfile);
  }
  ```
  
  ```
  $cmd="igprof-analyse --sqlite -d -v -g -r"
  $cmd ${mode} -b out1.gz --diff-mode out2.gz| sqlite3 diff.sql3
  ```

- Performance report formats
  - ascii text (flat file)
  - sqlite database files

- Demo for the web-navigable report
  [https://g4cpt.fnal.gov/g4p/oss_10.3.r04_lArTest_01/index_igprof.html](https://g4cpt.fnal.gov/g4p/oss_10.3.r04_lArTest_01/index_igprof.html)
# IgProf Pi FTFP BERT 5

## Counter: MEM_TOTAL, first 10

### Sorted by self cost

<table>
<thead>
<tr>
<th>Rank</th>
<th>Total %</th>
<th>Self</th>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>91.98%</td>
<td>34,869,868,357</td>
<td>859,743,283</td>
</tr>
<tr>
<td>33</td>
<td>7.25%</td>
<td>2,686,905,840</td>
<td>50,115</td>
</tr>
<tr>
<td>55</td>
<td>0.19%</td>
<td>70,819,848</td>
<td>8,645</td>
</tr>
<tr>
<td>60</td>
<td>0.12%</td>
<td>45,869,937</td>
<td>210</td>
</tr>
<tr>
<td>61</td>
<td>0.08%</td>
<td>30,122,856</td>
<td>190</td>
</tr>
<tr>
<td>67</td>
<td>0.04%</td>
<td>13,487,544</td>
<td>80,283</td>
</tr>
<tr>
<td>120</td>
<td>0.03%</td>
<td>12,453,440</td>
<td>311,336</td>
</tr>
<tr>
<td>159</td>
<td>0.02%</td>
<td>8,708,314</td>
<td>5,834</td>
</tr>
<tr>
<td>183</td>
<td>0.02%</td>
<td>6,534,144</td>
<td>2,070</td>
</tr>
<tr>
<td>222</td>
<td>0.02%</td>
<td>5,788,928</td>
<td>1,523</td>
</tr>
<tr>
<td>269</td>
<td>0.01%</td>
<td>4,321,280</td>
<td>250</td>
</tr>
<tr>
<td>277</td>
<td>0.01%</td>
<td>4,125,832</td>
<td>4,659</td>
</tr>
<tr>
<td>277</td>
<td>0.01%</td>
<td>4,023,395</td>
<td>405</td>
</tr>
<tr>
<td>293</td>
<td>0.01%</td>
<td>3,484,800</td>
<td>370</td>
</tr>
<tr>
<td>299</td>
<td>0.01%</td>
<td>3,237,888</td>
<td>3,162</td>
</tr>
<tr>
<td>306</td>
<td>0.01%</td>
<td>3,000,184</td>
<td>1</td>
</tr>
</tbody>
</table>

### Symbol name

- `std::basic_string<char, std::char_traits<char>, std::allocator<char> >::Rep::s_create(unsigned long, unsigned long, std::allocator<char>::deallocate12`
Hint from IgProf

• Q4: How to improve performance of this function? Avoid ...

```cpp
MaterialPropertyVector*
MaterialPropertiesTable::GetProperty(const char *key)
{
    // Returns a Material Property Vector corresponding to a key
    if (string(key) == "GROUPVEL") return SetGROUPVEL();

    MPTIterator i;
    i = MPT.find(string(key));
    if (i != MPT.end()) return i->second;
    return nullptr;
}
```

1. String comparison
2. String conversion
3. String search (find)
4. Race condition (map)
Introduction to OpenSpeedshop (OSS)

- Comprehensive performance analysis of sequential, multithreaded, and MPI applications
- Open source (the Krell institute, https://openspeedshop.org) and one of ASCR profiling tools
- The base functionality includes
  - Sampling experiment (light-weighted)
  - Support call stack analysis
  - Hardware performance (PAPI) counters
  - Multi-threaded, MPI profiling and tracing
  - Memory function tracing, I/O profiling and tracing, etc...
- Tested on a variety of Linux clusters and supports parallel hardware architectures (Intel MIC, NVIDIA CUDA) as well as HPC systems (Cray, Blue Gene)
OSS: Installation and Performance Measurement

- Installation: a typical build (with the version 2.2)

```bash
./install-tool --build-krell-root
  --krell-root-prefix ${install_dir}/krellroot_v2.2
  --with-openmpi /usr/local/openmpi-1.8.1
./install-tool --build-offline
  --openss-prefix ${install_dir}/opensspeedshop2.2
  --krell-root-prefix ${install_dir}/krellroot_v2.2
  --with-openmpi /usr/local/openmpi-1.8.1
```

- Running an experiment: unmodified binary instrumentation
  osspcsamp "lArTest lArBox.gdml profile.pi-5GeV" [frequency]

- Performance analysis: command-line (-cli) or GUI (-f)
  openss -cli lArTest-pcsamp.openss
# Demo: OSS Command-line Analysis (-cli)

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&gt; openss -cli</code></td>
<td>Open the CLI.</td>
</tr>
<tr>
<td><code>openss&gt;&gt; expcreate -f &quot;mutatee 2000&quot; pcsamp</code></td>
<td>Create an experiment using pcsamp with this application.</td>
</tr>
<tr>
<td><code>openss&gt;&gt; expgo</code></td>
<td>Run the experiment and create the database</td>
</tr>
<tr>
<td><code>openss&gt;&gt; expview</code></td>
<td>Display the default view of the performance data.</td>
</tr>
</tbody>
</table>

## help or help commands
- `expview`
- `expview -v statements`
- `expview -v loops`
- `expview -v linkedobjects`
- `expview -v calltrees,fullstack`
- `expview -m loadbalance`
- `expview -r <rank_num>`
- `expcompare -r 1 -r 2 -m time`
- `list -v metrics`
- `list -v src`

## list commands
- `list -v obj`
- `list -v ranks`
- `list -v hosts`
- `expview -m <metric>`
- `expview -v calltrees,fullstack <experiment type> <number>`
- `expview -v calltrees,fullstack usertime2`
- `expview <experiment-name><number>`
- `expview pcsamp2`
- `expview -v statements <experiment-name><number>`
OSS (GUI): Default View and Statistical Panel

![Image of OSS GUI with Toolbars and Top Functions highlighted]
Sampling Experiments in OSS

- pcsamp (periodic sampling of program counters)
  - low overhead overview of time distribution
- usertime (call path profiling)
  - inclusive and exclusive timing data
  - call paths, caller and callee relationships
- hwcsamp (periodic sampling hardware counters)
  - profiling of hardware counter events (PAPI events)
- pthreads (POSIX thread tracing)
- mem (memory tracing)
  - call paths of memory related function call events
  - aggregate and individual rank, thread, or processing times
- io (I/O tracing)
- Many other useful experiments
OSS: Measurement Overheads and Output Size

- pcsamp: exclusive time - insensitive to sampling frequency (default 100Hz)

<table>
<thead>
<tr>
<th>Frequency</th>
<th>Time(sec)</th>
<th>OverHead(%)</th>
<th>DB size(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>base:</td>
<td>52.20</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>50 Hz:</td>
<td>52.27</td>
<td>0.13</td>
<td>0.376832</td>
</tr>
<tr>
<td>100 Hz:</td>
<td>52.62</td>
<td>0.80</td>
<td>0.486400</td>
</tr>
<tr>
<td>200 Hz:</td>
<td>52.36</td>
<td>0.31</td>
<td>0.607232</td>
</tr>
<tr>
<td>500 Hz:</td>
<td>52.98</td>
<td>1.49</td>
<td>0.811008</td>
</tr>
<tr>
<td>1000 Hz:</td>
<td>52.65</td>
<td>0.86</td>
<td>0.971776</td>
</tr>
<tr>
<td>10000 Hz:</td>
<td>52.76</td>
<td>1.07</td>
<td>1.012736</td>
</tr>
</tbody>
</table>

- usertime: inclusive time and call paths – large overhead (default 35): similar overhead for hwcsamp

<table>
<thead>
<tr>
<th>Frequency</th>
<th>Time(sec)</th>
<th>OverHead(%)</th>
<th>DB size(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>base:</td>
<td>52.80</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>35 Hz:</td>
<td>53.89</td>
<td>2.06</td>
<td>1.087488</td>
</tr>
<tr>
<td>50 Hz:</td>
<td>54.33</td>
<td>2.90</td>
<td>1.430528</td>
</tr>
<tr>
<td>100 Hz:</td>
<td>56.21</td>
<td>6.46</td>
<td>2.355200</td>
</tr>
<tr>
<td>200 Hz:</td>
<td>60.25</td>
<td>14.11</td>
<td>4.208640</td>
</tr>
<tr>
<td>1000 Hz:</td>
<td>92.84</td>
<td>75.83</td>
<td>18.725888</td>
</tr>
</tbody>
</table>
Preliminary Performance Experiments with LArTest

- LArTest configuration
  - Beam: 5 GeV pi-
  - Step limit: 0.01 cm
  - Physics list: FTFP_BERT (uses standard EM)
  - 1000 events
- osspcsamp (100 Hz)
  - I/O (digitization) ON
  - Analysis ON
- ossusertime and osshwcsamp (35 Hz)
  - I/O (digitization) OFF
  - Analysis OFF
osspcsamp : Functions

- Exclusive CPU time - an overall performance view
osspcsamp: Statements (Line Numbers)

- Select statement level granularity
- List line numbers in program that took most of time
osspcsamp: Linked Objects

- The library in which the associated function is located (aggregated by shared objects)
ossusertime: Call Path (Functions)

- Function calls observed anywhere in the stack
- The inclusive time taken by the function and all its callees
ossusertime: Hot Call Path

- Relationship between caller and callee
- The paths through the application that take the most time
ossusertime: Hot Call (Source)

- Exclusive time on highlighted lines that indicate relatively high CPU times
Experiments with Hardware Counters

• Periodic sampling of hardware counters (hwcsamp)
• Supports both derived and non-derived PAPI presets
• Metrics for instructions, FLOPS, memory and resource patterns
Code Performance by Hardware Counter Metrics

• Derivatives: examples

<table>
<thead>
<tr>
<th>Hardware Counter Metrics Derivatives</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPC (Instruction/Cycle)</td>
<td>Large values suggest good balance with minimal stalls.</td>
</tr>
<tr>
<td>FPC (FLOPS/Cycle)</td>
<td>Large values for floating point intensive code suggests efficient CPU utilization</td>
</tr>
<tr>
<td>FMO (FLOPS/Memory Ops)</td>
<td>Good data locality, Computational Intensity</td>
</tr>
<tr>
<td>LPC (Loads/Cycle)</td>
<td>Useful for calculating FMO, may indicate good stride through arrays.</td>
</tr>
<tr>
<td>SPC (Stores/Cycle)</td>
<td>Useful for calculating FMO, may indicate good stride through arrays.</td>
</tr>
</tbody>
</table>

• LArTest (Overall): 5 GeV pi- (Intel Xeon X5650@2.67GHz)
  – IPC = 0.79 (relatively small)
  – FMO = 0.32
Other useful OSS Features

- Flexible analysis options *(GUI, command line, online)*
- Export report data in different formats *(text, cvs, chart)*
- Multi-threading capability
- Compare two experiments *(osscompare): examples*
  - two releases
  - two experiments with the different numbers of threads
- Call path analysis based on DB
- Experiments for parallel code *(MPI tracing)*
Demo: Example of Performance Profiling Report

- Monitor Geant4 part of performance changes for LAr-based detectors by
  - Beam energy/Particle type/Physics list
  - Geant4 (reference) release

https://g4cpt.fnal.gov/g4p/oss_10.3.r04_lArTest_01/index_sprof.html
https://g4cpt.fnal.gov/g4p/oss_10.3.r04_lArTest_01/index_igprof.html

OpenSpeedshop

Geant4.10.3.r04 lArTest

<table>
<thead>
<tr>
<th>Sample</th>
<th>Physics List</th>
<th>B-Field</th>
<th>Energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>e-</td>
<td>FTPP_BERT</td>
<td>OFF (0T)</td>
<td>1 GeV 5 GeV</td>
</tr>
<tr>
<td>e+</td>
<td>FTPP_BERT</td>
<td>OFF (0T)</td>
<td>1 GeV 5 GeV</td>
</tr>
<tr>
<td>mu-</td>
<td>FTPP_BERT</td>
<td>OFF (0T)</td>
<td>1 GeV 5 GeV</td>
</tr>
<tr>
<td>mu+</td>
<td>FTPP_BERT</td>
<td>OFF (0T)</td>
<td>1 GeV 5 GeV</td>
</tr>
<tr>
<td>p</td>
<td>FTPP_BERT</td>
<td>OFF (0T)</td>
<td>1 GeV 5 GeV</td>
</tr>
</tbody>
</table>

Memory Profiler/IgProf

Memory profiling reports

- MEM_LIVE: memory that has not been freed - snapshot of the heap, i.e. a heap profile.
- MEM_MAX: the largest single allocation by any function
- MEM_TOTAL: the total amount of memory allocated by any function - a snapshot of poor memory locality
- N: memory snapshot at the end of N-th event
- Diff(N-M): memory difference between N-th and M-th event - direct memory leakage
- End of Run: memory snapshot at the End of Run

Geant4.10.3.r04 lArTest B=4.0T

<table>
<thead>
<tr>
<th>Sample</th>
<th>Physics List</th>
<th>Energy</th>
<th>MEM_LIVE</th>
<th>MEM_MAX</th>
<th>MEM_TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>e-</td>
<td>FTPP_BERT</td>
<td>1 GeV 5 GeV</td>
<td>1 Diff(1001-1) 1001 End of Run</td>
<td>1 Diff(1001-1) 1001 End of Run</td>
<td>1 Diff(1001-1) 1001 End of Run</td>
</tr>
<tr>
<td>e+</td>
<td>FTPP_BERT</td>
<td>1 GeV 5 GeV</td>
<td>1 Diff(1001-1) 1001 End of Run</td>
<td>1 Diff(1001-1) 1001 End of Run</td>
<td>1 Diff(1001-1) 1001 End of Run</td>
</tr>
<tr>
<td>mu-</td>
<td>FTPP_BERT</td>
<td>1 GeV 5 GeV</td>
<td>1 Diff(1001-1) 1001 End of Run</td>
<td>1 Diff(1001-1) 1001 End of Run</td>
<td>1 Diff(1001-1) 1001 End of Run</td>
</tr>
<tr>
<td>mu+</td>
<td>FTPP_BERT</td>
<td>1 GeV 5 GeV</td>
<td>1 Diff(1001-1) 1001 End of Run</td>
<td>1 Diff(1001-1) 1001 End of Run</td>
<td>1 Diff(1001-1) 1001 End of Run</td>
</tr>
<tr>
<td>p</td>
<td>FTPP_BERT</td>
<td>1 GeV 5 GeV</td>
<td>1 Diff(1001-1) 1001 End of Run</td>
<td>1 Diff(1001-1) 1001 End of Run</td>
<td>1 Diff(1001-1) 1001 End of Run</td>
</tr>
</tbody>
</table>
Summary

• Performance profiling and analysis is an essential part of the software development cycle
  – Modern hardware architectures are demanding (parallelism)
  – HEP applications are big and complex
  – Profilers will help to identify critical parts of code, monitor changes of performance and provide opportunities of optimization

• Where you can start:
  – Try profiling your programs with basic tools
  – IgProf: http://igprof.org/index.html
  – OpenISpeedshop: https://openspeedshop.org/
  – HPCToolkits: http://hpctoolkit.org/index.html
  – TAU: http://www.cs.uoregon.edu/research/tau/home.php

• Above tools are quite suitable for continuous integration tests
Acronym

- CISC: Complicated instruction set computer
- RISC: Reduced instruction set computer
- ARM: Advanced RISC Machines
- CPI: Cycles per instruction
- IPC: Instructions per cycle
- MIPS: Million instructions per second
- FMO: Floating point operations per memory operation
- DRAM: Dynamic random-access memory
- ASCR: Advanced Scientific Computing Research
- MuMMI: Multiple Metrics Modeling Infrastructure
Answer sheet

- Q1:
- Q2:
- Q3:
- Q4: