HPC Lab

Session 4: Profiler

Sebastian Rettenberger, Chaulio Ferreira, Michael Bader

November 9, 2015
Profiler

Profiling allows you to learn where your program spent its time [...]. This information can show you which pieces of your program are slower than you expected, and might be candidates for rewriting to make your program execute faster.

Source: https://sourceware.org/binutils/docs/gprof/Introduction.html
**GNU gprof**

- Compile the program with profiling enabled:
  
  ```
  gcc -g -pg program.c -o program
  ```

- Execute the program and generate profile data:
  
  ```
  ./program
  ```
  
  (will generate a file `gmon.out`)

- Analyze the data with `gprof`:
  
  ```
  gprof program > output
  ```
Flat profile:

Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self</th>
<th>self</th>
<th>total</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>seconds</td>
<td>seconds</td>
<td>calls</td>
<td>us/call</td>
</tr>
<tr>
<td>21.75</td>
<td>0.05</td>
<td>0.05</td>
<td>5050000</td>
<td>0.01</td>
</tr>
<tr>
<td>13.05</td>
<td>0.08</td>
<td>0.03</td>
<td>5000</td>
<td>6.00</td>
</tr>
<tr>
<td>8.70</td>
<td>0.10</td>
<td>0.02</td>
<td>1010000</td>
<td>0.02</td>
</tr>
<tr>
<td>8.70</td>
<td>0.12</td>
<td>0.02</td>
<td>1010000</td>
<td>0.02</td>
</tr>
</tbody>
</table>

...
GNU gprof - Call tree

Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 4.35% of 0.23 seconds

index % time self children called name

... 0.02 0.00 5001/5001 main [1]
[12] 8.7 0.02 0.00 5001 writer::VtkWriter::write(float, float const*, float const*, unsigned int) [12]
0.00 0.00 5001/5001 writer::VtkWriter::generateFileName() [33]
0.00 0.00 5001/5002 std::operator|(std::_Ios_Openmode, std::_Ios_Openmode) [31]
...
On the cluster:

- `module load amplifier_xe`
- `GUI: amplxe-gui`
- `Command line tool: amplxe-cl`
Intel VTune Amplifier XE
Intel VTipne Amplifier XE

Elapsed Time: 15.269s
- Total Thread Count: 1
- Overhead Time: 0s
- Spin Time: 0s
- CPU Time: 5.876s
- Enabled Time: 0s

Top Hotspots
This section lists the most active handshakes in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function | CPU Time
--- | ---
- | 3.744s
- | 0.928s
- | 0.379s
- | 0.210s
- | 0.070s
- | 0.030s
- | 0.026s
- | 0.008s

CPU Usage Histogram
This histogram displays a percentage of the total time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the idle CPU usage...

Collection and Platform Info
This section provides information about this collection, including result set size and collection platform data.

Application Command Line: /home/rettenberger/Intel VTune Amplifier XE-2015.0.48.40-
Operating System: Ubuntu 18.04.4 LTS (Bionic Beaver) 64-bit (x86_64)
Computer Name: mos2login6
Result Size: 1.11M
Collection Start Time: 17:22:10-11/13/2015 UTC
Collection Stop Time: 17:22:10-11/13/2015 UTC

CPU
- Name: Unknown
- Frequency: 2.60 GHz
- Logical CPU Count: 24
Intel VTune Amplifier XE
Scalasca

- Open source project:
  Forschungszentrum Jülich,
  Technische Universität Darmstadt,
  German Research School for Simulation Sciences

On the Linux Cluster:

- `module load scalasca`
- Also loads:
  - `Scorep (Code instrumentation)``
  - `Cube (Visualization)`
Scalasca – instrumentation

- `scalasca -instrument [options] compiler ...`
  
  Installation on the Linux Cluster only works with the Intel compiler
  Custom installation for GCC possible

- Options:
  - `--mpp=mpi`
  - `--thread=omp`
  - `--nocompiler`
  - `--user`
  - ...

- Instrumented code generates a folder `scorep-*`
Cube
Cube
Score-P – Manual Instrumentation

- Option: --user
- Functions:

```c
#include <scorep/SCOREP_User.h>

void foo(x) {
    SCOREP_USER_REGION( "foo", SCOREP_USER_REGION_TYPE_FUNCTION )
    // Do something
}
```

- Regions:

```c
#include <scorep/SCOREP_User.h>

void foo() {
    SCOREP_USER_REGION_DEFINE( handle )
    // Do something
    SCOREP_USER_REGION_BEGIN( handle, "region",
                               SCOREP_USER_REGION_TYPE_COMMON )
    // Do something else
    SCOREP_USER_REGION_END( handle )
    // Do more
}
```
Score-P – Parameter-Based Profiling

```c
#include <scorep/SCOREP_User.h>

void foo(int64_t myint)
{
    SCOREP_USER_REGION_DEFINE( handle )
    SCOREP_USER_REGION_BEGIN( handle, "foo", SCOREP_USER_REGION_TYPE_COMMON )
    SCOREP_USER_PARAMETER_INT64( "myint", myint )
    // do something
    SCOREP_USER_REGION_END( handle )
}
```
Hardware Performance Counters

- Hardware counters are special registers
- Count events, e.g.
  - total instructions
  - cache misses
  - branch misses
  - ...
- Automatically incremented by the hardware
  → no instrumentation required
  → minimal overhead
Perf

- Linux tool/kernel module to read hardware counters
- Available on every modern Linux system
- Includes additional software counters:
  - page faults
  - context switches
  - I/O events
  - ...
## Perf

Performance counter stats for './a.out':

- **5.905492 task-clock** # 0.515 CPUs utilized
- **37 context-switches** # 0.006 M/sec
- **0 CPU-migrations** # 0.000 M/sec
- **356 page-faults** # 0.060 M/sec
- **13,656,215 cycles** # 2.312 GHz
  - [1.34%]
- **11,318,844 stalled-cycles-frontend** # 82.88% frontend cycles idle
  - [98.76%]
- **7,301,549 stalled-cycles-backend** # 53.47% backend cycles idle
- **9,840,841 instructions** # 0.72 insns per cycle
  - # 1.15 stalled cycles per insn
- **1,703,051 branches** # 288.384 M/sec
- **29,795 branch-misses** # 1.75% of all branches

0.011463205 seconds time elapsed
Performance counter stats for './a.out':

68,446 cache-references
17,377 cache-misses # 25.388 % of all cache refs
0.011199010 seconds time elapsed