HPC Lab

Session 4: Profiler

Carsten Uphoff, Chaulio Ferreira, Michael Bader
TUM – SCCS

December 5, 2016
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`
GNU gprof - Flat profile

Flat profile:

Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>time</th>
<th>seconds</th>
<th>self</th>
<th>calls</th>
<th>self/us/call</th>
<th>total/us/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>21.75</td>
<td>0.05</td>
<td>0.05</td>
<td>5050000</td>
<td>0.01</td>
<td>0.01</td>
<td>solver::vec2&lt;float&gt;::vec2(float)</td>
<td></td>
</tr>
<tr>
<td>13.05</td>
<td>0.08</td>
<td>0.03</td>
<td>5000</td>
<td>6.00</td>
<td>6.00</td>
<td>WavePropagation::updateUnknowns</td>
<td></td>
</tr>
<tr>
<td>8.70</td>
<td>0.10</td>
<td>0.02</td>
<td>1010000</td>
<td>0.02</td>
<td>0.03</td>
<td>solver::FWave&lt;float&gt;::f(solver::vec2&lt;float&gt;)</td>
<td></td>
</tr>
<tr>
<td>8.70</td>
<td>0.12</td>
<td>0.02</td>
<td>1010000</td>
<td>0.02</td>
<td>0.02</td>
<td>solver::matrix2x2&lt;float&gt;::matrix2x2()</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></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

<table>
<thead>
<tr>
<th>index</th>
<th>% time</th>
<th>self</th>
<th>children</th>
<th>called</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>[12]</td>
<td>8.7</td>
<td>0.02</td>
<td>0.00</td>
<td>5001</td>
<td>writer::VtkWriter::write(float, float const*, float const*, unsigned int) [12]</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>5001/5001</td>
<td></td>
<td>writer::VtkWriter::generateFileName() [33]</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>5001/5002</td>
<td></td>
<td>std::operator</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Intel VTune Amplifier XE

On the cluster:

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

Uphoff, Ferreira, Bader | HPC Lab | Session 4: Profiler | December 5, 2016
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
}
```
#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
Likwid-perfctr

- Linux tool/kernel module to read hardware counters
- Developed by Regionales Rechenzentrum Erlangen (RRZE)
- Includes additional software counters:
  - page faults
  - context switches
  - I/O events
  - ...
DGEMM with options:
Time/FLOPS with timeofday(): 1.020164e+10

Group 1: MEM

| Event                      | Counter | Core 1 |
|----------------------------+---------+---------|
| INSTR_RETIRED_ANY          | FIXC0   | 68146405143 |
| CPU_CLK_UNHALTED_CORE      | FIXC1   | 52138809008 |
| CPU_CLK_UNHALTED_REF       | FIXC2   | 41113421180 |

| Metric                     | Core 1 |
|----------------------------+---------|
| Memory read bandwidth [MBytes/s] | 1.014995e+04 |
| Memory read data volume [GBytes]   | 1.621715e+02 |
| Memory write bandwidth [MBytes/s]  | 4.532920e+01 |
| Memory write data volume [GBytes]  | 0.72425024 |
| Memory bandwidth [MBytes/s]       | 1.019528e+04 |
| Memory data volume [GBytes]        | 1.628957e+02 |
DGEMM with options: blocked packed vector
Time/FLOPS with timeofday(): 2.770782e+10

Group 1: MEM

<table>
<thead>
<tr>
<th>Event</th>
<th>Counter</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSTR_RETIRED_ANY</td>
<td>FIXC0</td>
<td>38479616403</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED_CORE</td>
<td>FIXC1</td>
<td>19245252158</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED_REF</td>
<td>FIXC2</td>
<td>15179510632</td>
</tr>
</tbody>
</table>

Memory read bandwidth [MBytes/s] | 5.727931e+03 |
Memory read data volume [GBytes] | 34.298501696 |
Memory write bandwidth [MBytes/s] | 3.039028e+02 |
Memory write data volume [GBytes] | 1.81975136 |
Memory bandwidth [MBytes/s] | 6.031834e+03 |
Memory data volume [GBytes] | 36.118253056 |
Things to keep in mind

- Measurements almost always create overhead.
- Check if a metric conforms with your performance model.
- Do not trust measurements blindly.