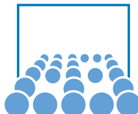
HPC Lab

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`

GNU gprof - Flat profile

```
 Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 21.75      0.05     0.05  5050000     0.01     0.01  solver::vec2<float>::vec2
 13.05      0.08     0.03     5000     6.00     6.00  WavePropagation::updateUn
  8.70      0.10     0.02  1010000     0.02     0.03  solver::FWave<float>::f(s
  8.70      0.12     0.02  1010000     0.02     0.02  solver::matrix2x2<float>:
  ...
```

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*, fl
               0.00    0.00    5001/5001        writer::VtkWriter::generateFileName() [33]
               0.00    0.00    5001/5002        std::operator|(std::_Ios_Openmode, std::_Ios
...
```

Intel VTune Amplifier XE

On the cluster:

- `module load amplifier_xe`
- GUI: `amplxe-gui`
- Command line tool: `amplxe-cl`

Intel VTune Amplifier XE

/home/hpc/pr63so/di56zem/intel/amplxe/SWE - Intel VTune Amplifier

Project Navigator

/home/hpc/pr63so/di56zem/intel/amplxe

HPC

SWE

Welcome | New Amplif...

Choose Analysis Type

Intel VTune Amplifier XE 2015

Analysis Type

- Algorithm Analysis
  - Basic Hotspots
  - Advanced Hotspots
  - Concurrency
  - Locks and Waits
- Microarchitecture Analysis
  - General Exploration
  - Bandwidth
  - CPU Specific Analysis
    - Intel Core 2 Processor Analysis
    - Nehalem / Westmere Analysis
    - Sandy Bridge Analysis
    - Haswell Analysis
      - TSX Exploration
- Knights Corner Platform Analysis
- Custom Analysis

Basic Hotspots

Copy

Identify your most time-consuming source code. This analysis type cannot be used to profile the system but must either launch an application/process or attach to one. This analysis type uses user-mode sampling and tracing collection. Press F1 for more details.

CPU sampling interval, ms: 10

Analyze user tasks

Details

Analyze GPU usage (Intel HD Graphics only):	No
CPU sampling interval, ms:	10
Collect CPU sampling data:	With stacks
Collect signalling API data:	No
Collect synchronization API data:	No
Collect I/O API data:	No
Analyze user tasks:	No
Analyze user synchronization:	No
Linux Ftrace events:	
Stack unwinding mode:	After collection
Stitch stacks:	Yes
Collect timeline data:	Yes
Collect sleep data:	No
Collect frequency data:	No

Start

Start Paused

Project Properties

Command Line...

Intel VTune Amplifier XE

/home/hpc/pr63so/di56zem/intel/amplxe/SWE - Intel VTune Amplifier <@mpp2-login6>

Project Navigator

/home/hpc/pr63so/di56zem/intel/amplxe
- HPC
- SWE

Welcome | r000hs

Basic Hotspots Hotspots by CPU Usage viewpoint (change)   Intel VTune Amplifier XE 2015

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree | Tasks and Frames

Elapsed Time: 15.269s

Total Thread Count:	1
Overhead Time:	0s
Spin Time:	0s
CPU Time:	5.070s
Paused Time:	0s

Top Hotspots

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

Function	CPU Time
std::ostream::flush	3.134s
std::ostream::_M_insert<double>	0.928s
std::basic_ofstream<char, std::char_traits<char>>::basic_ofstream	0.378s
std::basic_ofstream<char, std::char_traits<char>>::~basic_ofstream	0.210s
endl<char, std::char_traits<char> >	0.076s
[Others]	0.344s

CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU usage...

Elapsed Time: 15s, 12s, 9s, 5s, 3s, 0s
Average
Target Concurrency
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
Idle | Poor | Ok | Ideal | Over
Simultaneously Utilized Logical CPUs

Collection and Platform Info

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

Application Command Line:	/home/hpc/pr63so/di56zem/tmp/hpc4/session/tsunami/build/SWE1D -t 5000
Operating System:	3.0.101-0.46-default SUSE Linux Enterprise Server 11 (x86_64) VERSION = 11 PATCHLEVEL = 3
Computer Name:	mpp2-login6
Result Size:	1 MB
Collection start time:	17:21:44 10/11/2015 UTC
Collection stop time:	17:22:00 10/11/2015 UTC

CPU

Name:	Unknown
Frequency:	2.6 GHz
Logical CPU Count:	24

Intel VTune Amplifier XE

Basic Hotspots — Hotspots by CPU Usage viewpoint (change)

Intel VTune Amplifier XE 2015

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree | Tasks and Frames

Grouping: Call Stack

Function Stack	CPU Time: Total – Effective Time by Utilization	CPU Time: Total – Spin Time	CPU Time: Total – Overhead Time	CPU Time: Self – Effective Time by Utilization	CPU Time: Self – Spin Time	CPU Time: Self – Overhead Time	Mod..	Fu... (Full)	Sou.. File	Start Add..
Total	100.0%	0.0%	0.0%	0s	0s	0s				
__libc_start_main	100.0%	0.0%	0.0%	0s	0s	0s	libc ...	__li...		0x1 ..
main	100.0%	0.0%	0.0%	0s	0s	0s	SWE.	main		0x4 ..
writer::VtkWriter::write	94.1%	0.0%	0.0%	0.050s	0s	0s	SWE.	writ...		0x4 ..
WavePropagation::computeN	3.6%	0.0%	0.0%	0.010s	0s	0s	SWE.	Wav..		0x4 ..
solver::FWave<float>::com	3.4%	0.0%	0.0%	0.040s	0s	0s	SWE.	solv...		0x4 ..
tools::Logger::operator<<	1.3%	0.0%	0.0%	0s	0s	0s	SWE.	tool...		0x4 ..
tools::Logger::operator<<<flo	0.6%	0.0%	0.0%	0s	0s	0s	SWE.	tool...		0x4 ..
WavePropagation::setOutflov	0.2%	0.0%	0.0%	0.010s	0s	0s	SWE.	Wav..		0x4 ..
tools::Logger::operator<<<ch	0.2%	0.0%	0.0%	0s	0s	0s	SWE.	tool...		0x4 ..
Highlighted 1 row(s):	3.4%	0.0%	0.0%	0.040s	0s	0s				

Data Of Interest (CPU Metrics): 100.0% (0.040s of 0.040s)

SWE1D!solver::FWa...dates - [Unknown]
SWE1D!WavePropag...nown]:[Unknown]
SWE1D!main+0x41d...known]:[Unknown]
libc-2.11.3.so!__li...Unknown]:[Unknown]

SWE1D (TID: 2 — Thread — CPU Usage

No filters are applied. | Any Process | Any Thread | Any Module | Any Utilization | User functions + 1 | Inline Mode: on | Functions only

Scalasca

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

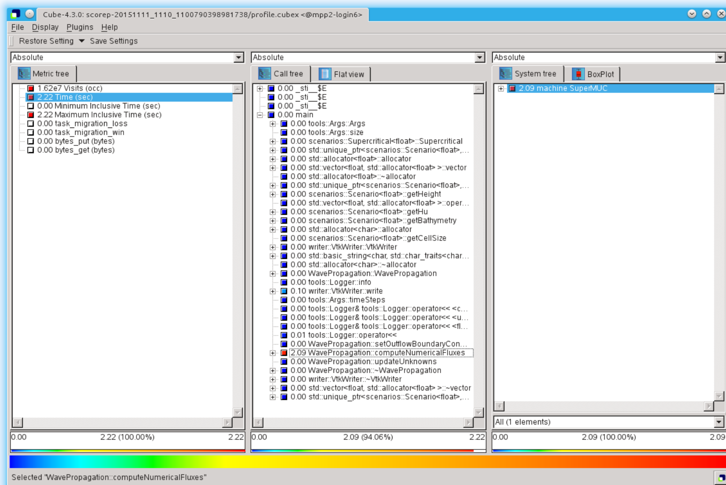
On the Linux Cluster:

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

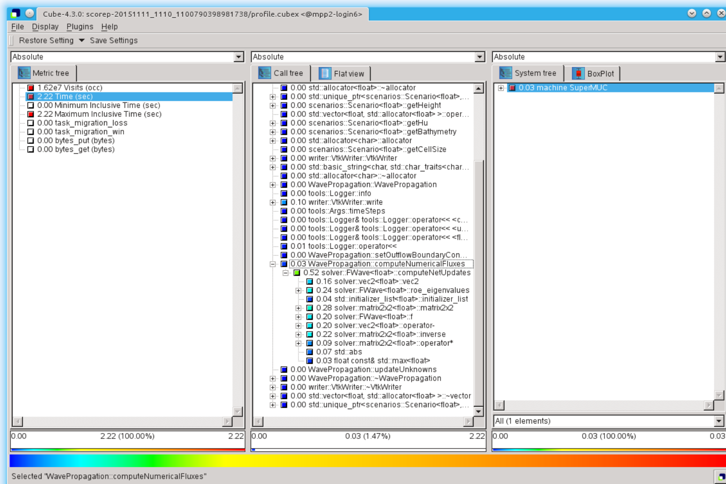
Scalasca – instrumentation

- `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-*`

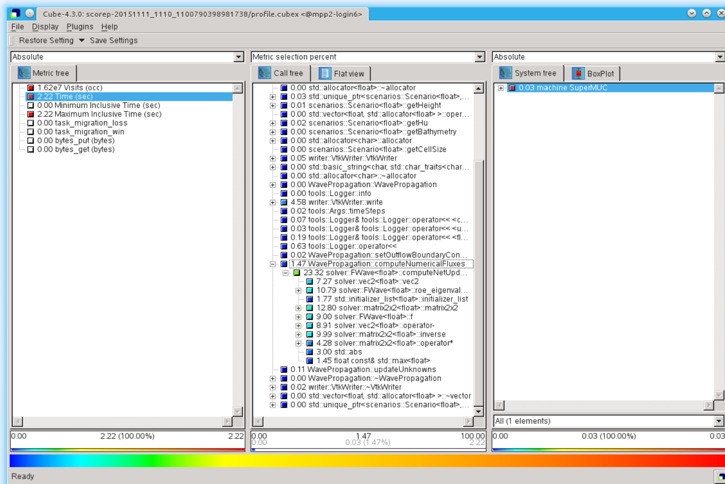
Cube



Cube



Cube



Score-P – Manual Instrumentation

- Option: `--user`
- Functions:

```
1 #include <scorep/SCOREP_User.h>
2
3 void foo(x) {
4   SCOREP_USER_REGION( "foo", SCOREP_USER_REGION_TYPE_FUNCTION )
5   // Do something
6 }
```

- Regions:

```
1 #include <scorep/SCOREP_User.h>
2 void foo() {
3   SCOREP_USER_REGION_DEFINE( handle )
4   // Do something
5   SCOREP_USER_REGION_BEGIN( handle, "region",
        SCOREP_USER_REGION_TYPE_COMMON )
6   // Do something else
7   SCOREP_USER_REGION_END( handle )
8   // Do more
9 }
```

Score-P – Parameter-Based Profiling

```
1  #include <scorep/SCOREP_User.h>
2
3  void foo(int64_t myint)
4  {
5    SCOREP_USER_REGION_DEFINE( handle )
6    SCOREP_USER_REGION_BEGIN( handle, "foo", SCOREP_USER_REGION_TYPE_COMMON
          )
7    SCOREP_USER_PARAMETER_INT64( "myint", myint )
8
9    // do something
10
11   SCOREP_USER_REGION_END( handle )
12 }
```

Hardware Performance Counters

- Hardware counters are special registers
- Count events, e.g.
  - total instructions
  - cache misses
  - branch misses
  - …
- Automatically incremented by the hardware
  $\rightarrow$ no instrumentation required
  $\rightarrow$ 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
```

Perf

```
1  Performance counter stats for './a.out':
2
3            68,446 cache-references
4            17,377 cache-misses            #   25.388 % of all cache refs
5
6       0.011199010 seconds time elapsed
```