Karlsruhe Institute of Technology

Performance Tools

Holger Obermaier

Steinbuch Centre for Computing (SCC)

- Optimization cycle
- Tool Test Cases
- Likwid Tools: Overview
- Likwid Tools: `likwid-topology`
- Likwid Tools: `likwid-bench`
- Compiler Optimization Report
- `/usr/bin/time`
- Application Performance Snapshot (APS)
- Likwid Tools: `likwid-perfctr`
- Likwid Tools: `likwid-perfctr` Marker API
- `perf` tools
- Intel Trace Analyzer and Collector (ITAC)
- References

Optimization cycle

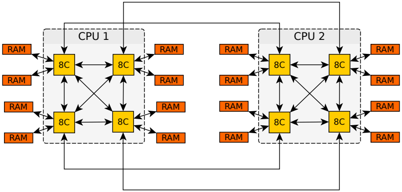
Karlsruhe Institute of Technology

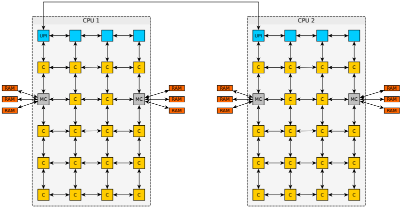
Current state of hardware development

- CPU cores do not get faster anymore
- More and more cores and nodes
- Multiple levels of caches try to hide memory latency

⇒ Optimizing code gets more complex

⇒ Support by performance tools is needed





Optimization cycle (2)

Iterative process

- Collect performance data
- Analyze data
  - Where is most of the time spent?
  - What is the expected performance?
  - Are cores evenly utilized?
  - Is memory access local?
  - Does communication limit performance?
- Fix problem
- Repeat until effort is no longer worth expected improvement

$\Rightarrow$ This talk focuses on performance data collection and analysis

Tool Test Cases

Benchmark *stream*

$$\begin{aligned} \text{Copy} \quad & c = a, \quad a, c \in \mathbb{R}^n \\ \text{Scale} \quad & b = \alpha c, \quad b, c \in \mathbb{R}^n, \quad \alpha \in \mathbb{R} \\ \text{Add} \quad & c = a + b, \quad a, b, c \in \mathbb{R}^n \\ \text{Triad} \quad & a = b + \alpha c, \quad a, b, c \in \mathbb{R}^n, \quad \alpha \in \mathbb{R} \end{aligned}$$

- $\mathcal{O}(n)$ memory operations, $\mathcal{O}(n)$ compute operations

$\Rightarrow$ Memory bandwidth bound

Benchmark *dgemm*

$$\text{Multiply} \quad C = A \cdot B, \qquad A, B, C \in \mathbb{R}^{n \times n}$$

- $\mathcal{O}(n^2)$ memory operations, $\mathcal{O}(n^3)$ compute operations

$\Rightarrow$ Floating point bound

Likwid Tools

- Collection of simple command line tools
- Hardware information:
  `likwid-topology`
- Micro benchmarks:
  `likwid-bench`
- Pinning:
  `likwid-pin`, `likwid-mpirun`
- Performance counters:
  `likwid-perfctr`

Likwid Tools: `likwid-topology`

- CPU topology (hardware threads, cores, sockets)
- Cache topology (location and size of caches)
- Cache properties (cache line size, associativity)
- NUMA topology (location and size of main memory)
- Get knowledge on how to bind your tasks, pin your threads

Example

- likwid-topology on Intel Xeon Broadwell
- likwid-topology cache topology on Intel Xeon Broadwell

Likwid Tools: `likwid-bench`

What is the maximum

- achievable memory bandwidth
- achievable cache bandwidth
- achievable computing power
- Vector (AVX, AVX2) computing power
- Fused multiply-add (FMA) computing power

Example

- likwid-bench on Intel Xeon Haswell

Compiler Optimization Report (Intel)

- Usage vectorization report

```
module add compiler/intel/18.0
icc ${OPT_FLAGS} \
    -qopt-report \
    -qopt-report-phase=vec \
    -qopt-report-stdout \
    ${SOURCE} -o ${OUTFILE}
```

Example

Intel optimization report: stream

Compiler Optimization report (GCC)

- Usage vectorization report

```
module add compiler/gnu/7
gcc ${OPT_FLAGS} \
    -fopt-info-vec \
    ${SOURCE} -o ${OUTFILE}
```

Example

GCC vectorization report: stream

`/usr/bin/time`

- No recompilation needed
  $\Rightarrow$ Use your existing binary
- Uses kernel resource usage info
- Report time consumption
  - time spent in user space
  - time spent in kernel space
  - elapsed time
- Report memory consumption
  - maximum resident size
  - Page faults
- Report IO operations

Example

Comparison *stream* serial/parallel execution with `time`

Application Performance Snapshot (APS)

- No recompilation needed
  $\Rightarrow$ Use your existing binary
- But: Best compatibility with Intel compiler and MPI
- Uses MPI library instrumentation
- Quick insight into
  - MPI
  - OpenMP
  - Memory access
  - Floating point
  - IO usage
- Text and HTML report

Application Performance Snapshot (APS) (2)

Karlsruhe Institute of Technology

- Usage serial or OpenMP binary

```
module add compiler/intel/18.0
source /opt/bwhpc/common/devel/aps/2018/apsvars.sh
aps ${BINARY}
```

Example

- APS: stream
- APS HTML report: stream
- APS: dgemm
- APS HTML report: dgemm

Application Performance Snapshot (3)

- Usage MPI binary

```
module add compiler/intel/18.0 \
           mpi/impi/2018-intel-18.0
source /opt/bwhpc/common/devel/aps/2018/apsvars.sh
mpirun aps ${BINARY}
```

Example

- APS: rank_league
- APS HTML report: rank_league

Likwid Tools: `likwid-perfctr`

- Measures total program performance
- No recompilation needed $\Rightarrow$ Use your existing binary
- Uses hardware performance *counters*
- Uses *sampling*
  - Low overhead
  - Only statistical results
- Performance groups simplify HW counters use
- Important performance groups

`FLOPS_AVX`	Packed AVX MFLOP/s
`MEM`	Main memory bandwidth
`NUMA`	Local and remote memory accesses

Likwid Tools: `likwid-perfctr` (2)

- Usage

```
likwid-perfctr -a   # Available performance groups
likwid-perfctr -H -group
    ${GROUP}   # Group information
likwid-perfctr -group ${GROUP} -C ${CPU_LIST}
    ${BINARY}   # Measure
```

Example

- likwid-perfctr: Performance group `NUMA` on benchmark stream
- likwid-perfctr: Performance group `FLOPS_AVX` on benchmark dgemm

Likwid Tools: `likwid-perfctr` Marker API

- Measure partial program performance
- Add likwid marker API to source code. Recompile.

`likwid_markerInit` Initialize likwid marker API

`likwid_markerThreadInit` Initialize each thread

`likwid_markerStartRegion` Start a measurement in named region

`likwid_markerStoptRegion` Stop a measurement in named region

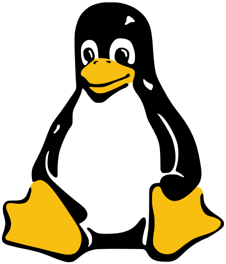
`likwid_markerClose` Close likwid marker API

Example

- Likwid marker API: stream
- Likwid marker API: dgemm

`perf` tools

- Part of Linux kernel
- No recompilation needed
  $\Rightarrow$ Use your existing binary
- Uses hardware performance *counters*
- Uses *sampling*
  - Low overhead
  - Only statistical results
- Find *hot spots*
  (functions or code regions)
- Record *call graph*
  (with compiler flag `-g`)



`perf` tools (2)

- Usage

```
perf list                    # available HW counters
perf stat    ${BINARY}       # profile w. HW counters
perf record  ${BINARY}       # measurement -> perf.data
perf report                  # Hot spot report
perf annotate                # Annotated assembler code
```

Example

- perf: dgemm
- perf: stream

Intel Trace Analyzer and Collector (ITAC)

- No recompilation needed
  $\Rightarrow$ Use your existing binary
- Uses *sampling*
  - Low overhead
  - Only statistical results
- Uses MPI library instrumentation
  - Collect non-statistical data
  - *Communication pattern*
  - *Message sizes*
- Can use compiler instrumentation
  - Can cause significant overhead
  - Collect non-statistical data
  - *Call graph*

Intel Trace Analyzer and Collector (ITAC) (2)

- Graphical tool shows
  - Event timeline
  - Quantitative timeline
  - Function profile
  - Message profile
- Usage

```
module add devel/itac/2018    # Prepare environment
mpirun -trace ${BINARY}       # Execute MPI program
traceanalyzer ${BINARY}.stf   # Analize data
```

Example:

- ITAC: MPI benchmark rank_league

References: Benchmarks

- DGEMM benchmark from Sandia National Laboratories
  http://www.nersc.gov/research-and-development/apex/apex-benchmarks/dgemm/
- Stream benchmark original version; John D. McCalpin
  https://www.cs.virginia.edu/stream/

References: Performance Tools

- Homepage: Application Performance Snapshot
  https://software.intel.com/sites/products/snapshots/application-snapshot/
- Homepage: Intel Trace Analyzer and Collector
  https://software.intel.com/en-us/intel-trace-analyzer
- Github-page: Likwid
  https://github.com/RRZE-HPC/likwid
- Homepage: Time
  https://directory.fsf.org/wiki/Time