Processor Specific Optimization

Part 2: Performance tools

J. Eitzinger

PATC LRZ, 18.3.2019
PRELUDE:
SCALABILITY 4 THE WIN!
Scalability Myth: Code scalability is the key issue

Lore 1
In a world of highly parallel computer architectures only highly scalable codes will survive

Lore 2
Single core performance no longer matters since we have so many of them and use scalable codes

Lore 3
Most codes suffer from load imbalance. So this is all you should care about.
Scalability Myth: Code scalability is the key issue

```c
!$OMP PARALLEL DO
do k = 1 , Nk
    do j = 1 , Nj; do i = 1 , Ni
        y(i,j,k) = b*( x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1) )
    enddo; enddo
enddo
!$OMP END PARALLEL DO
```

Changing only the compile options makes this code scalable on an 8-core chip:

- `-O3 -xAVX`
Scalability Myth: Code scalability is the key issue

```fortran
!$OMP PARALLEL DO
do k = 1 , Nk
    do j = 1 , Nj; do i = 1 , Ni
        y(i,j,k) = b*( x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1) )
    enddo; enddo
enddo
!$OMP END PARALLEL DO
```

Upper limit from simple performance model:
35 GB/s & 24 Byte/update

Single core/socket efficiency is key issue!

3D Stencil Update ("Jacobi")
PERFORMANCE TOOLS

- Overview
- likwid-topology
Tools for Node-level Performance Engineering

- Gather Node Information
  - hwloc, likwid-topology, likwid-powermeter

- Affinity control and data placement
  - OpenMP and MPI runtime environments, hwloc, numactl, likwid-pin

- Runtime Profiling
  - Compilers, gprof, HPC Toolkit, …

- Performance Profilers
  - Intel Vtune™, likwid-perfctr, PAPI based tools, Linux perf, …

- Microbenchmarking
  - STREAM, likwid-bench, Imbench
How do we figure out the node topology?

**LIKWID** tool suite:

Like
I
Knew
What
I’m
Doing

Open source tool collection (developed at RRZE):
https://github.com/RRZE-HPC/likwid


http://arxiv.org/abs/1004.4431
Likwid Tool Suite

- Command line tools for Linux:
  - easy to install
  - works with standard linux kernel
  - simple to use
  - supports Intel and AMD CPUs

- Current tools:
  - **likwid-topology**: Print thread and cache topology
  - **likwid-pin**: Pin threaded application without touching code
  - **likwid-perfctr**: Measure performance counters
  - **likwid-powermeter**: Query turbo mode steps. Measure ETS.
  - **likwid-bench**: Low-level bandwidth benchmark generator tool
Output of likwid-topology -g
on one node of Intel Haswell-EP

CPU name: Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz
CPU type: Intel Xeon Haswell EN/EP/EX processor
CPU stepping: 2

Hardware Thread Topology

Sockets: 2
Cores per socket: 14
Threads per core: 2

<table>
<thead>
<tr>
<th>HWThread</th>
<th>Thread</th>
<th>Core</th>
<th>Socket</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>*</td>
</tr>
<tr>
<td>44</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>*</td>
</tr>
</tbody>
</table>

Socket 0: ( 0 28 1 29 2 30 3 31 4 32 5 33 6 34 7 35 8 36 9 37 10 38 11 39 12 40 13 41 )
Socket 1: ( 14 42 15 43 16 44 17 45 18 46 19 47 20 48 21 49 22 50 23 51 24 52 25 53 26 54 27 55 )

Cache Topology

Level: 1
Size: 32 kB
Cache groups: ( 0 28 ) ( 1 29 ) ( 2 30 ) ( 3 31 ) ( 4 32 ) ( 5 33 ) ( 6 34 ) ( 7 35 ) ( 8 36 ) ( 9 37 ) ( 10 38 ) ( 11 39 ) ( 12 40 ) ( 13 41 ) ( 14 42 ) ( 15 43 ) ( 16 44 ) ( 17 45 ) ( 18 46 ) ( 19 47 ) ( 20 48 ) ( 21 49 ) ( 22 50 ) ( 23 51 ) ( 24 52 ) ( 25 53 ) ( 26 54 ) ( 27 55 )

Level: 2
Size: 256 kB
Cache groups: ( 0 28 ) ( 1 29 ) ( 2 30 ) ( 3 31 ) ( 4 32 ) ( 5 33 ) ( 6 34 ) ( 7 35 ) ( 8 36 ) ( 9 37 ) ( 10 38 ) ( 11 39 ) ( 12 40 ) ( 13 41 ) ( 14 42 ) ( 15 43 ) ( 16 44 ) ( 17 45 ) ( 18 46 ) ( 19 47 ) ( 20 48 ) ( 21 49 ) ( 22 50 ) ( 23 51 ) ( 24 52 ) ( 25 53 ) ( 26 54 ) ( 27 55 )

Level: 3
Size: 17 MB
Cache groups: ( 0 28 1 29 2 30 3 31 4 32 5 33 6 34 ) ( 7 35 8 36 9 37 10 38 11 39 12 40 13 41 ) ( 14 42 15 43 16 44 17 45 18 46 19 47 20 48 ) ( 21 49 22 50 23 51 24 52 25 53 26 54 27 55 )
Output of likwid-topology continued

********************************************************************************
NUMA Topology
********************************************************************************
NUMA domains: 4
--------------------------------------------------------------------------------
Domain: 0
Processors: ( 0 28 1 29 2 30 3 31 4 32 5 33 6 34 )
Distances: 10 21 31 31
Free memory: 13292.9 MB
Total memory: 15941.7 MB
--------------------------------------------------------------------------------
Domain: 1
Processors: ( 7 35 8 36 9 37 10 38 11 39 12 40 13 41 )
Distances: 21 10 31 31
Free memory: 13514 MB
Total memory: 16126.4 MB
--------------------------------------------------------------------------------
Domain: 2
Processors: ( 14 42 15 43 16 44 17 45 18 46 19 47 20 48 )
Distances: 31 31 10 21
Free memory: 15025.6 MB
Total memory: 16126.4 MB
--------------------------------------------------------------------------------
Domain: 3
Processors: ( 21 49 22 50 23 51 24 52 25 53 26 54 27 55 )
Distances: 31 31 21 10
Free memory: 15488.9 MB
Total memory: 16126 MB
Output of likwid-topology continued

Graphical Topology

Cluster on die mode and SMT enabled!
ENFORCING THREAD/PROCESS-CORE AFFINITY UNDER THE LINUX OS

- Standard tools and OS affinity facilities under program control
- likwid-pin
- OpenMP 4
Example: STREAM benchmark on 16-core Sandy Bridge: Anarchy vs. thread pinning

- There are several reasons for caring about affinity:
  - Eliminating performance variation
  - Making use of architectural features
  - Avoiding resource contention
  - Benchmark how code reacts to variations

No pinning

Pinning (physical cores first, first socket first)
More thread/Process-core affinity ("pinning") options

- Highly OS-dependent system calls
  - But available on all systems
    - Linux: `sched_setaffinity()`
    - Windows: `SetThreadAffinityMask()`

- Hwloc project (http://www.open-mpi.de/projects/hwloc/)

- Support for “semi-automatic” pinning in some compilers/environments
  - All modern compilers with OpenMP support
  - Generic Linux: `taskset`, `numactl`, `likwid-pin` (see below)
  - OpenMP 4.0

- Affinity awareness in MPI libraries
  - SGI MPT
  - OpenMPI
  - Intel MPI
  - …
Likwid-pin

Overview

- Pins processes and threads to specific cores without touching code
- Directly supports pthreads, gcc OpenMP, Intel OpenMP
- Based on combination of wrapper tool together with overloaded pthread library → binary must be dynamically linked!
- Can also be used as a superior replacement for taskset
- Supports logical core numbering within a node

Usage examples:

- `likwid-pin -c 0-3,4,6 ./myApp parameters`
- `likwid-pin -c S0:0-7 ./myApp parameters`
- `likwid-pin -c N:0-15 ./myApp parameters`
LIKWID terminology

Thread group syntax

- The OS numbers all processors (hardware threads) on a node
- The numbering is enforced at boot time by the BIOS
- LIKWID introduces thread groups consisting of processors sharing a topological entity (e.g. socket or shared cache)
- A thread group is defined by a single character + index

Example for likwid-pin:
likwid-pin -c S1:0-3,6,7 ./a.out

Thread group expression may be chained with @:
likwid-pin -c S0:0-3@S1:0-3 ./a.out

Alternative expression based syntax:
likwid-pin -c E:S0:4:2:2 ./a.out
E:<thread domain>:<num threads>:<chunk size>:<stride>

Xeon Phi: likwid-pin -c E:N:60:2:4 ./a.out
Likwid

Currently available thread domains

Possible unit prefixes

N node

S socket

M NUMA domain

C outer level cache group

Default if –c is not specified!
Likwid-pin
Example: Intel OpenMP

Running the STREAM benchmark with likwid-pin:

```
$ likwid-pin -c S0:0-3 ./stream
[likwid-pin] Main PID -> core 0 - OK
----------------------------------------------
Double precision appears to have 16 digits of accuracy
Assuming 8 bytes per DOUBLE PRECISION word
----------------------------------------------
Array size = 20000000
Offset = 32
The total memory requirement is 457 MB
You are running each test 10 times
--
The *best* time for each test is used
*EXCLUDING* the first and last iterations
[pthread wrapper] [pthread wrapper] PIN_MASK: 0->1 1->2 2->3
[pthread wrapper] SKIP MASK: 0x1
  threadid 140370139711232 -> SKIP
  threadid 140370117211968 -> core 1 - OK
  threadid 140370113013632 -> core 2 - OK
  threadid 140369974597568 -> core 3 - OK
[... rest of STREAM output omitted ...]
```

Main PID always pinned
Skip shepherd thread
Pin all spawned threads in turn
A place consists of one or more processors. Pinning is done on the level of places.
Free migration of the threads on a place between the processors of that place.

- **OMP_PLACES=threads**
  - Each place corresponds to the single processor of a single hardware thread (hyper-thread)

- **OMP_PLACES=cores**
  - Each place corresponds to the processors (one or more hardware threads) of a single core

- **OMP_PLACES=sockets**
  - Each place corresponds to the processors of a single socket (consisting of all hardware threads of one or more cores)

- **OMP_PLACES=abstract_name(num_places)**
  - In general, the number of places may be explicitly defined

- Or use explicit numbering, e.g. 8 places, each consisting of 4 processors:
  - **OMP_PLACES="{0,1,2,3},{4,5,6,7},{8,9,10,11}, ... {28,29,30,31}"**
  - **OMP_PLACES="{0:4},{4:4},{8:4}, ... {28:4}"**
  - **OMP_PLACES="{0:4}:8:4"**
OMP_PROC_BIND variable / proc_bind() clause

- Determines how places are used for pinning:

<table>
<thead>
<tr>
<th>OMP_PROC_BIND</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>FALSE</td>
<td>Affinity disabled</td>
</tr>
<tr>
<td>TRUE</td>
<td>Affinity enabled, implementation defined strategy</td>
</tr>
<tr>
<td>CLOSE</td>
<td>Threads bind to consecutive places</td>
</tr>
<tr>
<td>SPREAD</td>
<td>Threads are evenly scattered among places</td>
</tr>
<tr>
<td>MASTER</td>
<td>Threads bind to the same place as the master thread that was running before the parallel region was entered</td>
</tr>
</tbody>
</table>

- If there are more threads than places, consecutive threads are put into individual places ("balanced")
Some simple OMP_PLACES examples

- Intel Xeon w/ SMT, 2x10 cores, 1 thread per physical core, fill 1 socket
  
  OMP_NUM_THREADS=10  
  OMP_PLACES=cores  
  OMP_PROC_BIND=close

- Intel Xeon Phi with 72 cores, 32 cores to be used, 2 threads per physical core
  
  OMP_NUM_THREADS=64  
  OMP_PLACES=cores(32)  
  OMP_PROC_BIND=close

- Intel Xeon, 2 sockets, 4 threads per socket (no binding within socket!)
  
  OMP_NUM_THREADS=8  
  OMP_PLACES=sockets  
  OMP_PROC_BIND=close

- Intel Xeon, 2 sockets, 4 threads per socket, binding to cores
  
  OMP_NUM_THREADS=8  
  OMP_PLACES=cores  
  OMP_PROC_BIND=spread
HARDWARE PERFORMANCE METRICS AND FREQUENCY CONTROL

- likwid-powermeter
- likwid-setFrequencies
- likwid-perfctr
Which clock speed steps are there?

**likwid-powermeter**

Uses the Intel RAPL (Running average power limit) interface (Sandy Bridge++)

```bash
$ likwid-powermeter -i
```

---

<table>
<thead>
<tr>
<th>CPU name</th>
<th>Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU type</td>
<td>Intel Xeon Haswell EN/EP/EX processor</td>
</tr>
<tr>
<td>CPU clock</td>
<td>2.30 GHz</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Base clock</th>
<th>2300.00 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimal clock</td>
<td>1200.00 MHz</td>
</tr>
</tbody>
</table>

**Turbo Boost Steps:**

- C0 3300.00 MHz
- C1 3300.00 MHz
- C2 3100.00 MHz
- C3 3000.00 MHz
- C4 2900.00 MHz
- [...] C13 2800.00 MHz

---

**Info for RAPL domain PKG:**

- Thermal Spec Power: 120 Watt
- Minimum Power: 70 Watt
- Maximum Power: 120 Watt
- Maximum Time Window: 46848 micro sec

---

**Info for RAPL domain DRAM:**

- Thermal Spec Power: 21.5 Watt
- Minimum Power: 5.75 Watt
- Maximum Power: 21.5 Watt
- Maximum Time Window: 44896 micro sec

---

**likwid-powermeter** can also measure energy consumption, but **likwid-perfctr** can do it better (see later)

---

Note: AVX code on HSW+ may execute even slower than base freq.
Setting the clock frequency

- The “Turbo Mode” feature makes reliable benchmarking harder
  - CPU can change clock speed at its own discretion
- Clock speed reduction may save a lot of energy

- So how do we set the clock speed? \( \rightarrow \) LIKWID to the rescue!

```bash
$ likwid-setFrequencies -l
Available frequencies:
1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2 2.1 2.2 2.3 2.301
$ likwid-setFrequencies -p
Current CPU frequencies:
CPU 0: governor performance min/cur/max 2.3/2.301/2.301 GHz Turbo 1
CPU 1: governor performance min/cur/max 2.3/2.301/2.301 GHz Turbo 1
CPU 2: governor performance min/cur/max 2.3/2.301/2.301 GHz Turbo 1
CPU 3: governor performance min/cur/max 2.3/2.301/2.301 GHz Turbo 1
[...]$ likwid-setFrequencies -f 2.0
$```

Turbo mode
Starting with Intel Haswell, the Uncore (L3, memory controller, UPI) sits in its own clock domain.

$ likwid-setFrequencies -p
[...]
CPU 68: governor performance min/cur/max 2.3/2.301/2.301 GHz Turbo 1
CPU 69: governor performance min/cur/max 2.3/2.301/2.301 GHz Turbo 1
CPU 70: governor performance min/cur/max 2.3/2.301/2.301 GHz Turbo 1
CPU 71: governor performance min/cur/max 2.3/2.301/2.301 GHz Turbo 1

Current Uncore frequencies:
Socket 0: min/max 1.2/3.0 GHz
Socket 1: min/max 1.2/3.0 GHz

$ likwid-setFrequencies --umin 2.3 --umax 2.3

- Uncore has considerable impact on power consumption
  - J. Hofmann et al.: On the accuracy and usefulness of analytic energy models for contemporary multicore processors. Proc. ISC High Performance 2018. DOI: 10.1007/978-3-319-92040-5_2
likwid-perfctr
Basic approach to performance analysis

1. Runtime profile / Call graph (gprof): Where are the hot spots?
2. Instrument hot spots (prepare for detailed measurement)
3. Find performance signatures

Possible signatures:
- Bandwidth saturation
- Instruction throughput limitation (real or language-induced)
- Latency impact (irregular data access, high branch ratio)
- Load imbalance
- ccNUMA issues (data access across ccNUMA domains)
- Pathologic cases (false cacheline sharing, expensive operations)

Goal: Come up with educated guess about a performance-limiting motif (Performance Pattern)
Probing performance behavior

- How do we find out about the performance properties and requirements of a parallel code?
  - Profiling via advanced tools is often overkill
- A coarse overview is often sufficient
  - `likwid-perfctr`
    - Simple end-to-end measurement of hardware performance metrics
    - “Marker” API for starting/stopping counters
    - Multiple measurement region support
    - Preconfigured and extensible metric groups, list with `likwid-perfctr -a`

Metric groups:
- **BRANCH**: Branch prediction miss rate/ratio
- **CACHE**: Data cache miss rate/ratio
- **CLOCK**: Clock of cores
- **DATA**: Load to store ratio
- **FLOPS_DP**: Double Precision MFlops/s
- **FLOPS_SP**: Single Precision MFlops/s
- **FLOPS_X87**: X87 MFlops/s
- **L2**: L2 cache bandwidth in MBytes/s
- **L2CACHE**: L2 cache miss rate/ratio
- **L3**: L3 cache bandwidth in MBytes/s
- **L3CACHE**: L3 cache miss rate/ratio
- **MEM**: Main memory bandwidth in MBytes/s
- **TLB**: TLB miss rate/ratio
likwid-perfctr

Example usage with preconfigured metric group

$ likwid-perfctr -g L2 -C S1:0-3 ./a.out

CPU name: Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz [...]

<<< PROGRAM OUTPUT >>>>

Group 1: L2

<table>
<thead>
<tr>
<th>Event</th>
<th>Counter</th>
<th>Core 14</th>
<th>Core 15</th>
<th>Core 16</th>
<th>Core 17</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSTR_RETIRED_ANY</td>
<td>FIXC0</td>
<td>1298031144</td>
<td>1965945005</td>
<td>1854182290</td>
<td>1862521357</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED_CORE</td>
<td>FIXC1</td>
<td>2353698512</td>
<td>2894134935</td>
<td>2894645261</td>
<td>2895023739</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED_REF</td>
<td>FIXC2</td>
<td>2057044629</td>
<td>2534405765</td>
<td>2535218217</td>
<td>2535560434</td>
</tr>
<tr>
<td>L1D_REPLACEMENT</td>
<td>PMC0</td>
<td>212900444</td>
<td>200544877</td>
<td>200389272</td>
<td>200387671</td>
</tr>
<tr>
<td>L2_TRANS_L1D_WR</td>
<td>PMC1</td>
<td>112464863</td>
<td>99931184</td>
<td>99982371</td>
<td>99976697</td>
</tr>
<tr>
<td>ICACHE_MISSES</td>
<td>PMC2</td>
<td>21265</td>
<td>26233</td>
<td>12646</td>
<td>12363</td>
</tr>
</tbody>
</table>

[... statistics output omitted ...]

<table>
<thead>
<tr>
<th>Metric</th>
<th>Core 14</th>
<th>Core 15</th>
<th>Core 16</th>
<th>Core 17</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime (RDTSC) [s]</td>
<td>1.1314</td>
<td>1.1314</td>
<td>1.1314</td>
<td>1.1314</td>
</tr>
<tr>
<td>Runtime unhalted [s]</td>
<td>1.0234</td>
<td>1.2583</td>
<td>1.2586</td>
<td>1.2587</td>
</tr>
<tr>
<td>Clock [MHz]</td>
<td>2631.6699</td>
<td>2626.4367</td>
<td>2626.0579</td>
<td>2626.0468</td>
</tr>
<tr>
<td>CPI</td>
<td>1.8133</td>
<td>1.4721</td>
<td>1.5611</td>
<td>1.5544</td>
</tr>
<tr>
<td>L2D load bandwidth [MBytes/s]</td>
<td>12042.7388</td>
<td>11343.8446</td>
<td>11335.0428</td>
<td>11334.9523</td>
</tr>
<tr>
<td>L2D evict bandwidth [MBytes/s]</td>
<td>6361.5883</td>
<td>5652.6192</td>
<td>5655.5146</td>
<td>5655.1937</td>
</tr>
<tr>
<td>L2 bandwidth [MBytes/s]</td>
<td>18405.5299</td>
<td>16997.9477</td>
<td>16991.2728</td>
<td>16990.8453</td>
</tr>
</tbody>
</table>
likwid-perfctr
Marker API (C/C++ and Fortran)

- A marker API is available to restrict measurements to code regions
- The API only turns counters on/off. The configuration of the counters is still done by likwid-perfctr
- Multiple named region support, accumulation over multiple calls
- Inclusive and overlapping regions allowed

```
#include <likwid.h>

... LIKWID_MARKER_INIT; // must be called from serial region
#pragma omp parallel
{
  LIKWID_MARKER_THREADINIT; // only reqd. if measuring multiple threads
}
... LIKWID_MARKER_START("Compute");
... LIKWID_MARKER_STOP("Compute");
... LIKWID_MARKER_START("Postprocess");
... LIKWID_MARKER_STOP("Postprocess");
... LIKWID_MARKER_CLOSE; // must be called from serial region
```

- **Activate macros with** `-DLIKWID_PERFMON`
- **Run** `likwid-perfctr` **with** `-m` **switch to enable marking**
- **See** [https://github.com/RRZE-HPC/likwid/wiki/TutorialMarkerF90](https://github.com/RRZE-HPC/likwid/wiki/TutorialMarkerF90) **for Fortran example**
likwid-perfctr

Compiling, linking, and running with the marker API

Compile:
cc -I /path/to/likwid.h -DLIKWID_PERFMON -c program.c

Link:
cc program.o -L /path/to/liblikwid -llikwid

Run:
likwid-perfctr -C <MASK> -g <GROUP> -m ./a.out

→ One separate block of output for every marked region
→ Caveat: marker API can cause overhead; do not call too frequently!
Summary of hardware performance monitoring

- Useful only if you know what you are looking for
  - PM bears potential of acquiring massive amounts of data for nothing!

- Resource-based metrics are most useful
  - Cache lines transferred, work executed, loads/stores, cycles
  - Instructions, CPI, cache misses may be misleading

- Caveat: Processor work $\neq$ user work
  - Waiting time in libraries (OpenMP, MPI) may incur lots of instructions
  - $\rightarrow$ distorted application characteristic

- Another very useful application of PM: validating performance models!
  - Roofline is data centric $\rightarrow$ measure data volume through memory