NHR FAU

Multicore Performance and Tools

Tools for Node-level Performance Engineering

- **Node Information**
  *`/proc/cpuinfo`, `numactl`, `hwloc`, **`likwid-topology`**, `likwid-powermeter`*

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

- **Runtime Profiling**
  Compilers, `gprof`, `perf`, HPC Toolkit, Intel Amplifier, …

- **Performance Analysis**
  *Intel VTune, **`likwid-perfctr`**, PAPI-based tools, HPC Toolkit, Score-P Tools, Linux `perf`*

- **Microbenchmarking**
  *STREAM, **`likwid-bench`**, lmbench, uarch-bench*

LIKWID performance tools

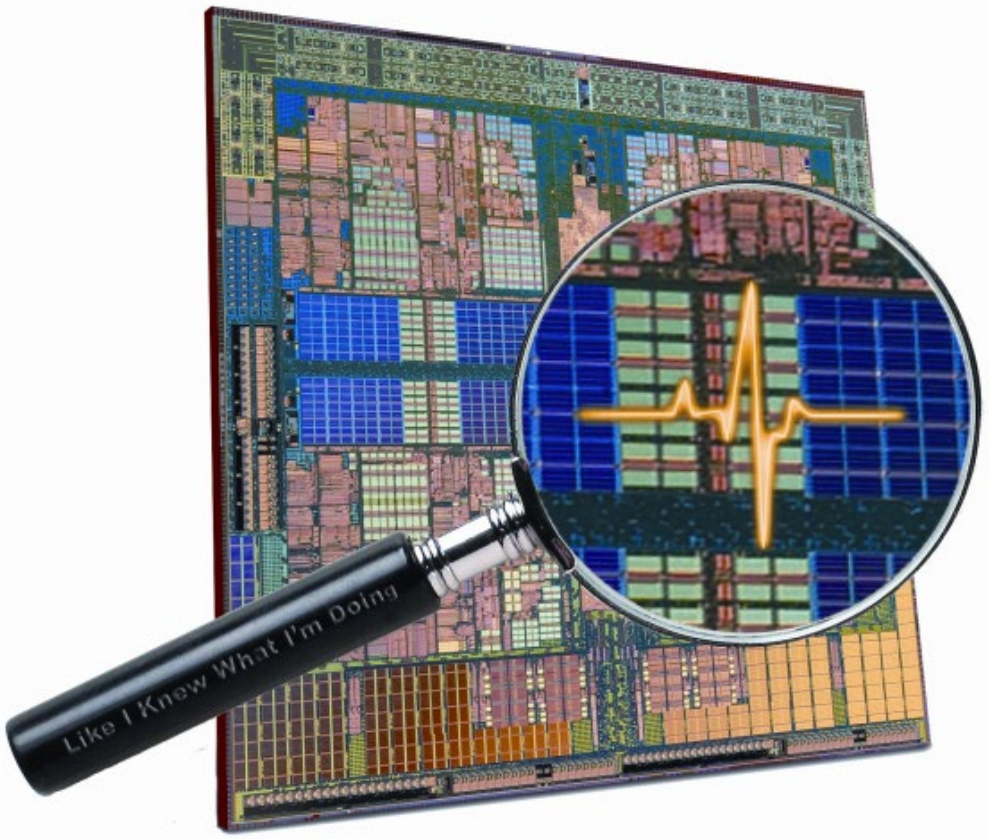
LIKWID tool suite:

Like
I
Knew
What
I'm
Doing

https://youtu.be/6uFl1HPq-88

Open source tool collection
(developed at RRZE):

https://github.com/RRZE-HPC/likwid



J. Treibig, G. Hager, G. Wellein: *LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments.* PSTI2010, Sep 13-16, 2010, San Diego, CA. DOI: 10.1109/ICPPW.2010.38

LIKWID Tool Suite

- Command line tools for Linux:
  - easy to install
  - works with standard Linux kernel
  - simple and clear to use
  - supports most X86 CPUs

  (also ARMv8, POWER9 and Nvidia GPUs)

- Current tools:

likwid-topology - Print thread and cache topology

likwid-pin - Pin threaded application without touching code

likwid-perfctr - Measure performance counters

likwid-powermeter - Measure energy consumption

likwid-bench - Microbenchmarking tool and environment

... some more

optional

Reporting topology

likwid-topology

https://youtu.be/mxMWjNe73SI

Overview `likwid-topology`

- Present hardware topology
  - HW thread topology
  - Cache topology
  - NUMA topology
  - GPU topology (if built with a GPU backend)
  - Graphical (ASCII) output

- Uses hwloc under the hood
  - Fallback to own detection if hwloc fails

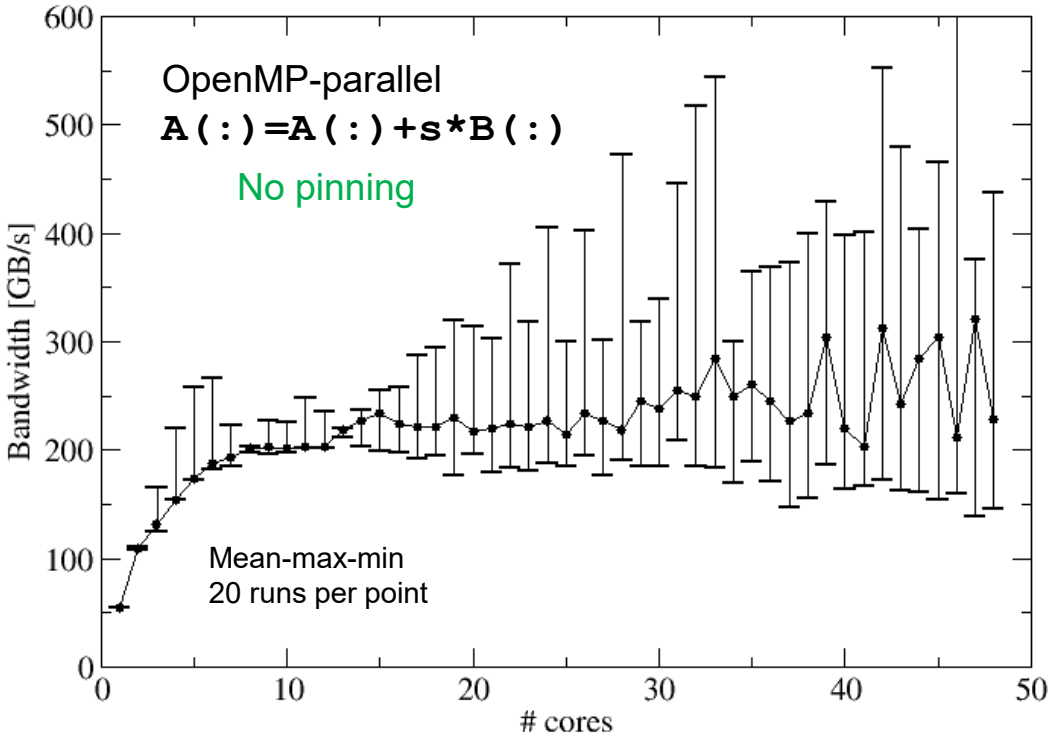
Enforcing thread/process affinity under the Linux OS

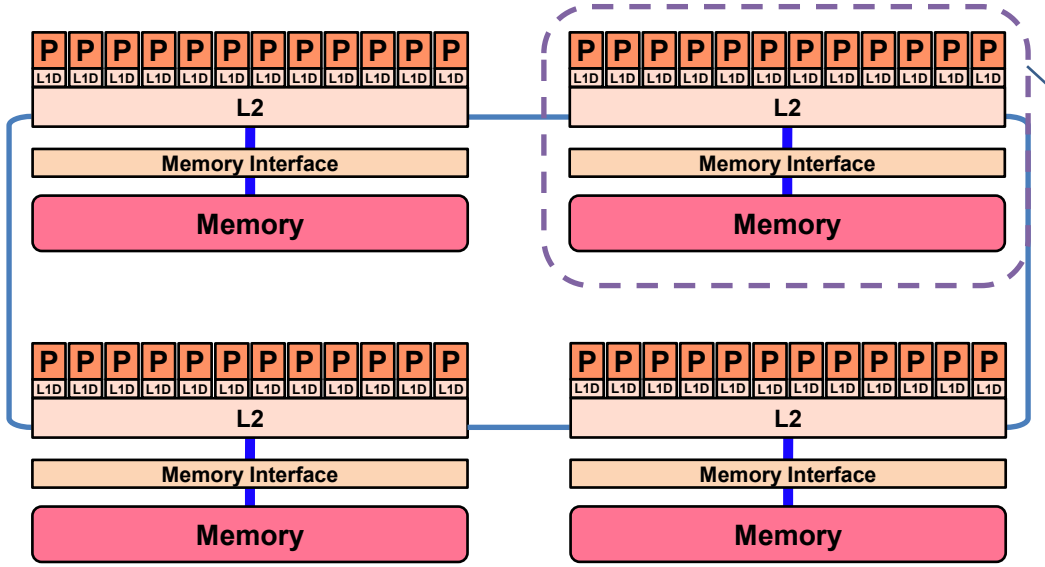
likwid-pin

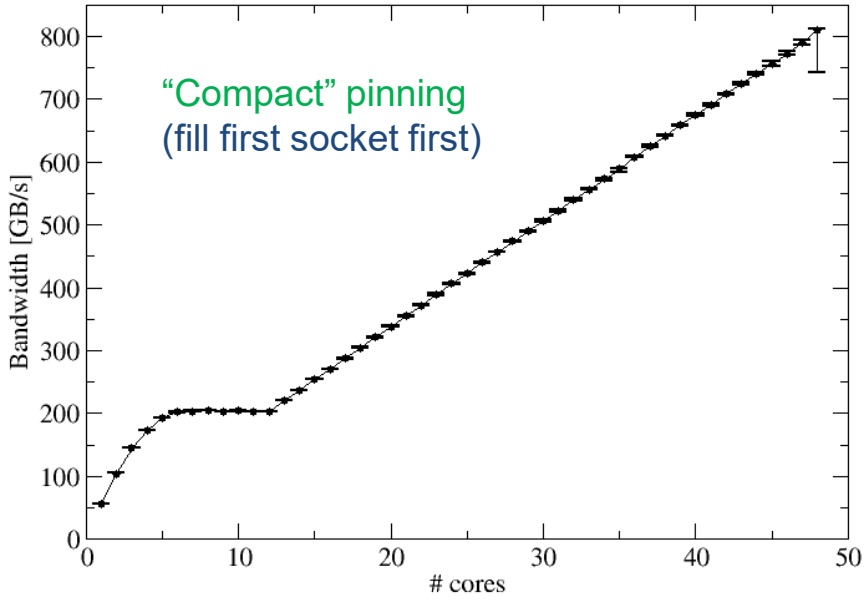
https://youtu.be/PSJKNQaqwB0

DAXPY test on A64FX

Anarchy vs. thread pinning



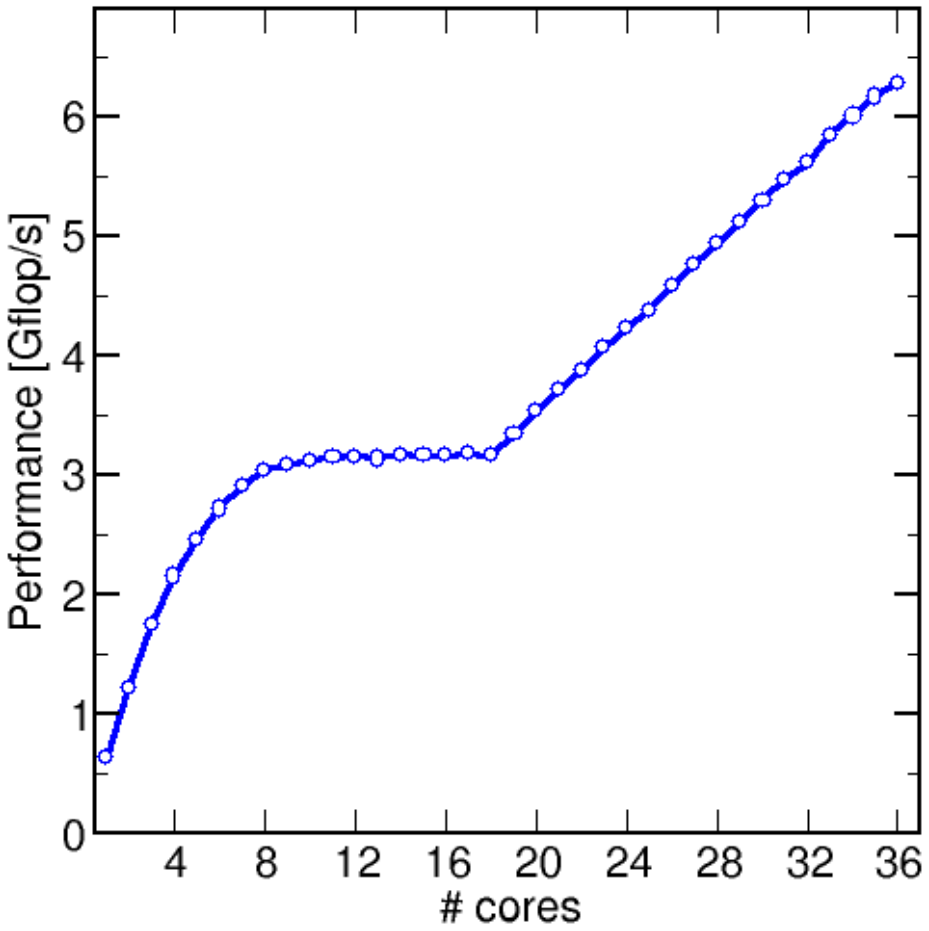




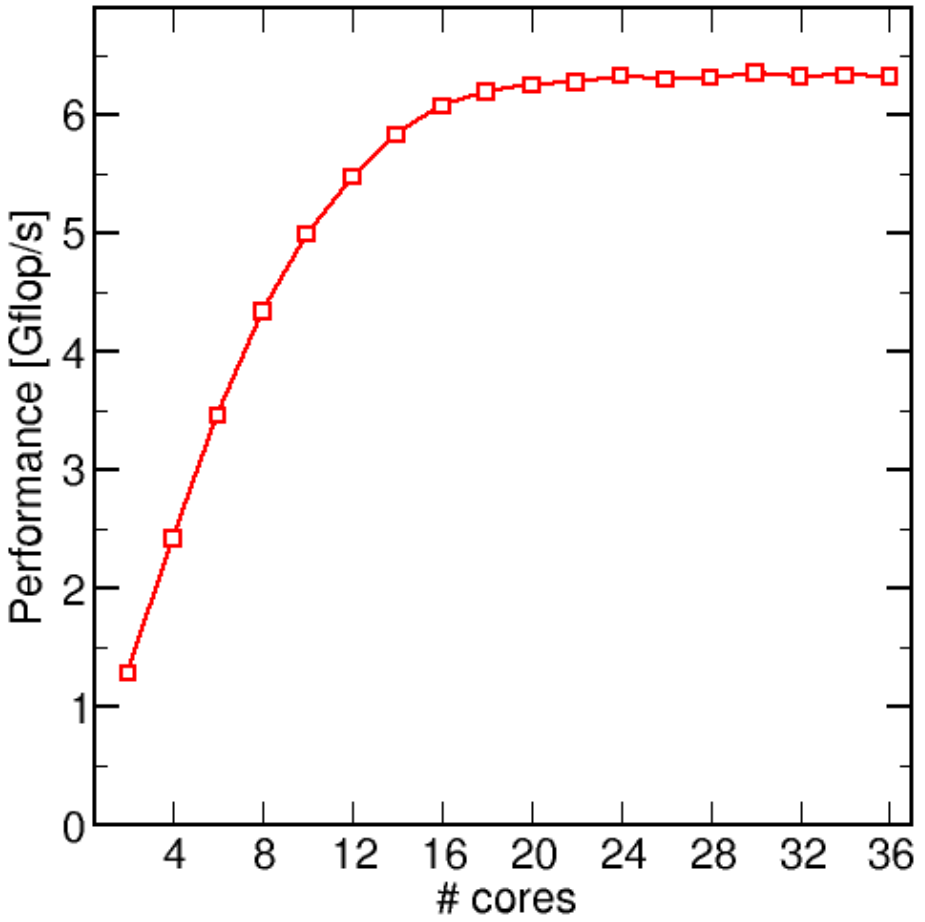
There are several reasons for caring about affinity:

- Eliminating performance variation
- Making use of architectural features
- Avoiding resource contention

Vector triad on 2x 18-core node: Compact vs. spread pinning



Filling cores from left to right
("compact" pinning)



Filling both sockets simultaneously
("scattered" or "spread" pinning)

More thread/process affinity ("pinning") options

- Highly OS-dependent system calls but available on all systems
  - Linux: **`sched_setaffinity()`**
  - Windows: **`SetThreadAffinityMask()`**
  - OS X: **`thread_policy_set()`**
- Hwloc project (http://www.open-mpi.de/projects/hwloc/)
- Support for "semi-automatic" pinning
  - All modern compilers with OpenMP support OpenMP 4.0 (**`OMP_PLACES`**, **`OMP_PROC_BIND`**)
  - CPUset reduction utils: **`taskset`** or **`numactl`**
  - Job scheduler like **`SLURM`**
- Affinity awareness in MPI libraries (OpenMPI, Intel MPI, …)

Overview `likwid-pin`

- 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!
- Supports logical core numbering within topological entities (thread domains)

- Simple usage with physical (kernel) core IDs:

```
$ likwid-pin -c 0-3,4,6  ./myApp parameters
$ OMP_NUM_THREADS=4 likwid-pin -c 0-9 ./myApp params
```

- Simple usage with logical core IDs (“thread groups”):

```
$ likwid-pin -c S0:0-7 ./myApp params
$ likwid-pin –c C1:0-2 ./myApp params
```

LIKWID terminology: Thread group syntax

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

Physical HW threads first!

```
+-----------------------------------+
| +------+ +------+ +------+ +------+ |
| |  0  4| |  1  5| | 2  6 | | 3  7 | |
| +------+ +------+ +------+ +------+ |
+-----------------------------------+
```

- Example for likwid-pin:

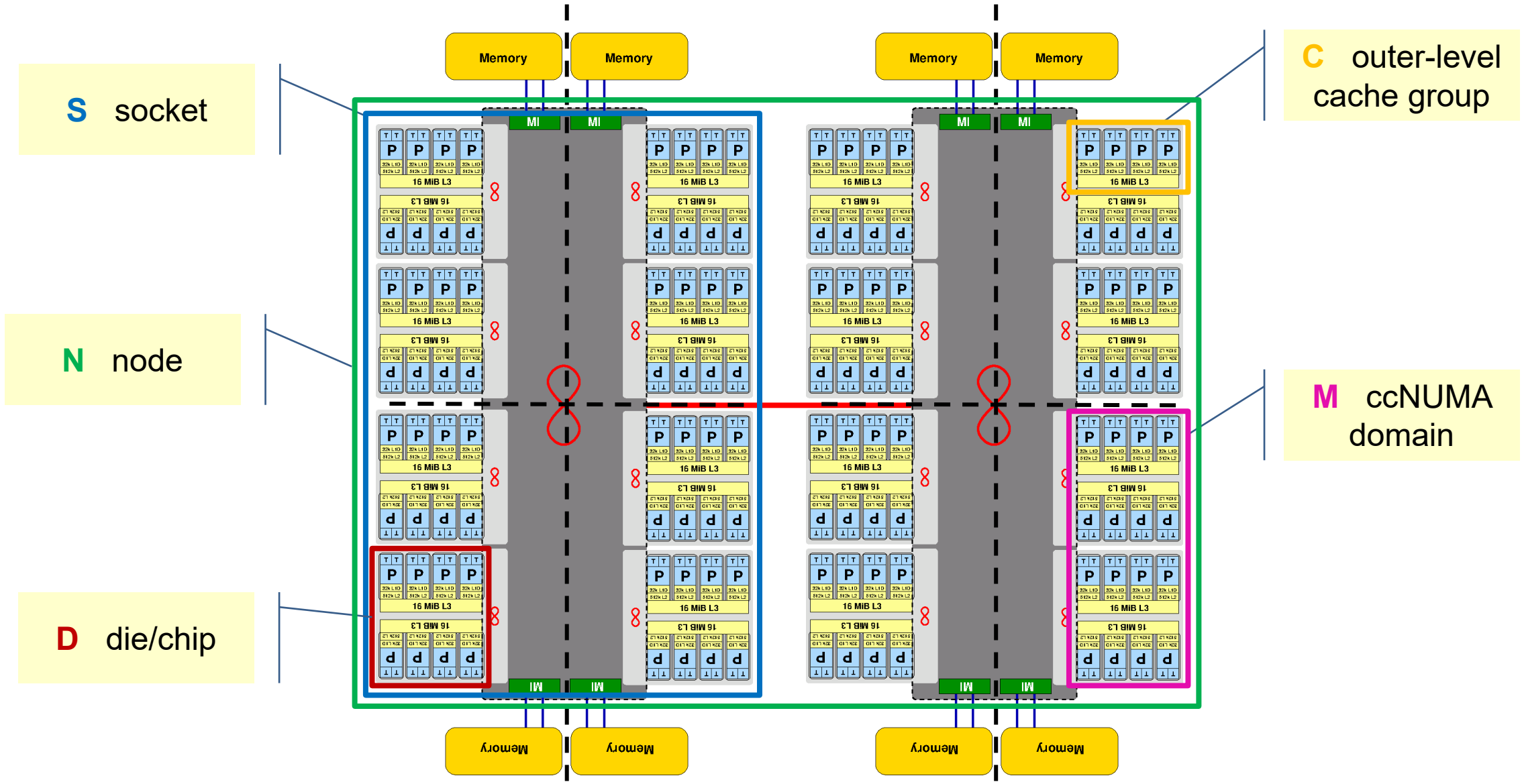
```
$ likwid-pin -c S0:0-3 ./a.out
```

- Thread group expressions may be chained with `@`:

```
$ likwid-pin -c S0:0-2@S1:0-2 ./a.out
```

```
+-----------------------------------++-----------------------------------+
| +------+ +------+ +------+ +------+ || +------+ +------+ +------+ +------+ |
| | 0  8 | | 1  9 | | 2 10 | | 3 11 | || | 4 12 | | 5 13 | | 6 14 | | 7 15 | |
| +------+ +------+ +------+ +------+ || +------+ +------+ +------+ +------+ |
+-----------------------------------++-----------------------------------+
```

Available thread domains/unit prefixes (as of LIKWID 5.3+)



Example: `likwid-pin` with Intel OpenMP

Running the STREAM benchmark with `likwid-pin`:

```
$ likwid-pin -c S0:0-3 ./stream
-------------------------------------------
 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] MAIN -> 0
[pthread wrapper] PIN_MASK: 0->1  1->2  2->3
[pthread wrapper] SKIP MASK: 0x0
        threadid 47308666070912 -> core 1 - OK
        threadid 47308670273536 -> core 2 - OK
        threadid 47308674476160 -> core 3 - OK

  [... rest of STREAM output omitted ...]
```

Main PID always pinned

Pin all spawned threads in turn

OMP_PLACES and Thread Affinity

optional

Processor: smallest entity able to run a thread or task (hardware thread)

Place: one or more processors → thread pinning is done place by place

Free migration of the threads on a place between the processors of that place.

abstract name

OMP_PLACES	Place ==
threads	Hardware thread (hyper-thread)
cores	All HW threads of a single core
sockets	All HW threads of a socket
abstract_name(num_places)	Restrict # of places available

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"`

<lower-bound>:<number of entries>[:<stride>]

Caveat: Actual behavior is implementation defined!

OMP_PROC_BIND variable / proc_bind() clause

optional

Determines how places are used for pinning:

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

If there are more threads than places, consecutive threads are put into individual places (“balanced”)

Some simple OMP_PLACES examples

optional

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
```

Always prefer abstract places instead of HW thread IDs!

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    # spread will also do
```

Intel Xeon, 2 sockets, 4 threads per socket (no binding within socket!)

```
OMP_NUM_THREADS=8
OMP_PLACES=sockets
OMP_PROC_BIND=close    # spread will also do
```

Intel Xeon, 2 sockets, 4 threads per socket, binding to cores

```
OMP_NUM_THREADS=8
OMP_PLACES=cores
OMP_PROC_BIND=spread
```

MPI startup and hybrid pinning: `likwid-mpirun`

- How do you manage affinity with MPI or hybrid MPI/threading?
- In the long run a unified standard is needed (will probably not happen)
- Till then, likwid-mpirun provides a portable/flexible solution
- The examples here are for Intel MPI/OpenMP programs, but are also applicable to other threading models

Pure MPI:

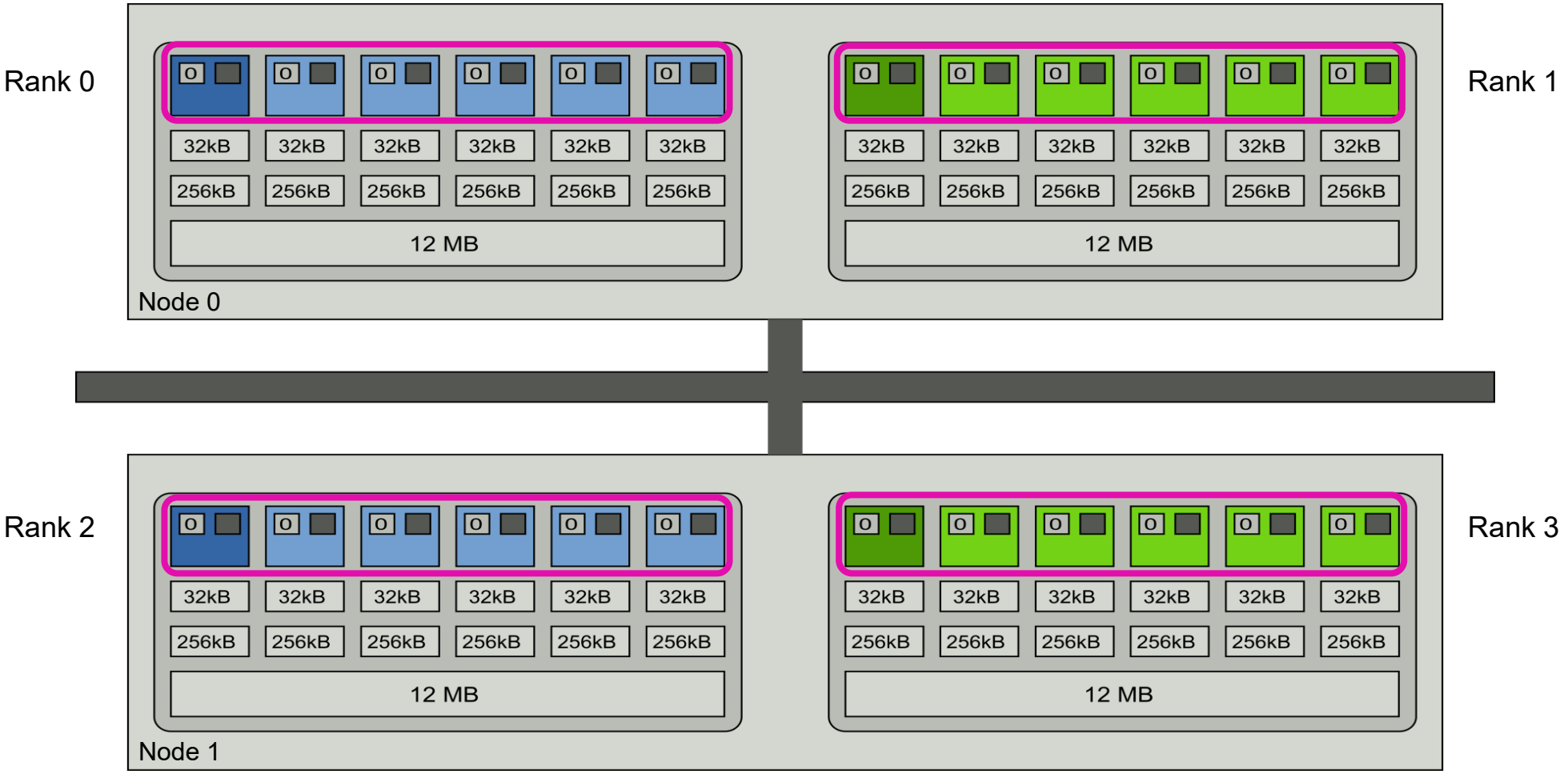
```
$ likwid-mpirun -np 16 -nperdomain S:2 ./a.out
```

Hybrid:

```
$ likwid-mpirun -np 16 -pin S0:0,1_S1:0,1 ./a.out
$ likwid-mpirun -np 16 -nperdomain S:1 -t 2 ./a.out
```

likwid-mpirun 1 MPI process per socket

```
$ likwid-mpirun -np 4 -pin S0:0-5_S1:0-5 ./a.out
```



Intel MPI+compiler:

```
OMP_NUM_THREADS=6 mpirun -ppn 2 -np 4 \
     -env I_MPI_PIN_DOMAIN socket -env KMP_AFFINITY scatter ./a.out
```

NHR FAU

optional

Clock speed under the Linux OS

Turbo steps and likwid-powermeter

likwid-setFrequencies

Which clock speed steps are there?

optional

Uses the Intel RAPL interface (Sandy Bridge++)

```
$ likwid-powermeter -i
--------------------------------------------------------------------------------
CPU name:       Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz
CPU type:       Intel Xeon Haswell EN/EP/EX processor
CPU clock:      2.30 GHz
--------------------------------------------------------------------------------
Base clock:     2300.00 MHz
Minimal clock:  1200.00 MHz
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
--------------------------------------------------------------------------------
```

Note: AVX code on HSW+ may execute even slower than base freq.

```
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)

Setting the clock frequency

optional

- 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?

  → LIKWID to the rescue!

You have to ask your cluster administrators if frequency control is possible, e.g., by using slurm functionality.

```
$ 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
$ 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    # min=max=2.0
[...]
$ likwid-setFrequencies -turbo 0 # turbo off
```

Turbo mode

Uncore clock frequency

optional

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.: *An analysis of core- and chip-level architectural features in four generations of Intel server processors*. Proc. ISC High Performance 2017. DOI: 10.1007/978-3-319-58667-0_16.

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

Fixing the clock frequency with Slurm

- Fritz allows you to set the clock speed when running your binary
  - Use **`srun`** with the **`--cpu-freq=<MIN>-<MAX>:<governor>`** option as a wrapper to your binary
  - Clock frequency is specified in kHz here (god knows why…)

```
user@f0772:~$ srun --cpu-freq=2000000-2000000:performance ./a.out
```

- On DINE2: `--cpu-freq=2100000-2100000:performance`

NHR FAU

Measuring hardware performance counters

likwid-perfctr

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

Operating modes:

- Wrapper
- Stethoscope
- Timeline
- Marker API

Preconfigured and extensible metric groups, list with
`likwid-perfctr -a`

```
BRANCH: Branch prediction miss rate/ratio
CACHE: Data cache miss rate/ratio
CLOCK: Clock frequency of cores
DATA: Load to store ratio
FLOPS_DP: Double Precision MFlops/s
FLOPS_SP: Single Precision 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
ENERGY: Power and energy consumption
```

likwid-perfctr wrapper mode

```
$ likwid-perfctr -g L2 -C S1:0-3 ./a.out
--------------------------------------------------------------------------------
CPU name:  Intel(R) Xeon(R) Platinum 8360Y CPU @ 2.40GHz[…]
--------------------------------------------------------------------------------
<<<< PROGRAM OUTPUT >>>>
--------------------------------------------------------------------------------
Group 1: L2
+----------------------+---------+-------------+-------------+-------------+-------------+
|         Event        | Counter | HWThread 36 | HWThread 37 | HWThread 38 | HWThread 39 |
+----------------------+---------+-------------+-------------+-------------+-------------+
|   INSTR_RETIRED_ANY  |  FIXC0  |  1409713380 |  1393263859 |  1394342491 |  1388917034 |
| CPU_CLK_UNHALTED_CORE|  FIXC1  |  2095261718 |  2088036330 |  2075539220 |  2058287996 |
|  CPU_CLK_UNHALTED_REF|  FIXC2  |  2103679392 |  2121235200 |  2100479808 |  2075658144 |
|   TOPDOWN_SLOTS      |  FIXC3  | 10476308590 | 10440181650 | 10377696100 | 10291439980 |
|   L1D_REPLACEMENT    |   PMC0  |   142720376 |   142481840 |   142482162 |   142434419 |
|   L2_TRANS_L1D_WB    |   PMC1  |    54986306 |    54864382 |    54868339 |    54815549 |
| ICACHE_64B_IFTAG_MISS|   PMC2  |      381869 |        2094 |        7399 |        7718 |
+----------------------+---------+-------------+-------------+-------------+-------------+
[… statistics output omitted …]
+--------------------------------+-------------+-------------+-------------+-------------+
|             Metric             | HWThread 36 | HWThread 37 | HWThread 38 | HWThread 39 |
+--------------------------------+-------------+-------------+-------------+-------------+
|       Runtime (RDTSC) [s]      |      1.0092 |      1.0092 |      1.0092 |      1.0092 |
|      Runtime unhalted [s]      |      0.8751 |      0.8721 |      0.8669 |      0.8597 |
|           Clock [MHz]          |   2384.7406 |   2356.8484 |   2365.8917 |   2374.2844 |
|               CPI              |      1.4863 |      1.4987 |      1.4885 |      1.4819 |
|  L2D load bandwidth [MBytes/s] |   9050.5857 |   9035.4589 |   9035.4794 |   9032.4518 |
|  L2D load data volume [GBytes] |      9.1341 |      9.1188 |      9.1189 |      9.1158 |
| L2D evict bandwidth [MBytes/s] |   3486.9462 |   3479.2144 |   3479.4653 |   3476.1177 |
| L2D evict data volume [GBytes] |      3.5191 |      3.5113 |      3.5116 |      3.5082 |
|    L2 bandwidth [MBytes/s]     |  12561.7480 |  12514.8061 |  12515.4139 |  12509.0589 |
|    L2 data volume [GBytes]     |     12.6777 |     12.6303 |     12.6309 |     12.6245 |
+--------------------------------+-------------+-------------+-------------+-------------+
```

Always measured for Intel CPUs

Configured metrics (this group)

Derived metrics

`likwid-perfctr` with MarkerAPI

- The MarkerAPI can restrict measurements to code regions
- The API only reads counters.
  The configuration of the counters is still done by `likwid-perfctr`
- Multiple named regions allowed, accumulation over multiple calls
- Inclusive and overlapping regions allowed

- Caveat: Marker API can cause significant overhead; do not call too frequently!

```
#include <likwid-marker.h>

LIKWID_MARKER_INIT; // must be called from serial region
. . .
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
```

likwid-perfctr with MarkerAPI: OpenMP code (C)

```c
#include <likwid-marker.h>

int main(...) {
  LIKWID_MARKER_INIT;
  #pragma omp parallel
  {
    LIKWID_MARKER_REGISTER("MatrixAssembly");
  }
  ...
  #pragma omp parallel
  {
    LIKWID_MARKER_START("MatrixAssembly");
    #pragma omp for
    for(int i=0; i<N; ++i) { /* Loop */ }
    LIKWID_MARKER_STOP("MatrixAssembly");
  }
  ...
  LIKWID_MARKER_CLOSE;
}
```

Optional: Prepare data structures (reduced overhead on 1st marker call)

Call markers in parallel region if data should be taken on all threads

https://github.com/RRZE-HPC/likwid/wiki/TutorialMarkerC

likwid-perfctr with MarkerAPI: OpenMP code (Fortran)

```
program p
  use likwid
  call likwid_markerInit
  !$omp parallel
    call likwid_markerRegisterRegion(“MatrixAssembly”)
  !$omp end parallel
  ...
  !$omp parallel
    call likwid_markerStartRegion(“MatrixAssembly”)
    !$omp do
    do i=1,N
     ! Loop
    enddo
    !$omp end do
    call likwid_markerStopRegion(“MatrixAssembly”)
  !$omp end parallel
  ...
  call likwid_markerClose
end program p
```

Optional: Prepare data structures (reduced overhead on 1st marker call)

Call markers in parallel region if data should be taken on all threads

https://github.com/RRZE-HPC/likwid/wiki/TutorialMarkerF90

`likwid-perfctr` with MarkerAPI: source code transformations

```
#pragma omp parallel for
    <loop>
```

```
#pragma omp parallel
{
    LIKWID_MARKER_START("Compute");
    #pragma omp for
    <loop>
    LIKWID_MARKER_STOP("Compute");
}
```

```
some_parallel_f()
```

```
#pragma omp parallel
{
    LIKWID_MARKER_START("foo");
}
some_parallel_f()
#pragma omp parallel
{
    LIKWID_MARKER_STOP("foo");
}
```

Compiling, linking, and running with marker API

Compile:

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

Activate LIKWID macros (C only)

Link:

```
cc -L /path/to/liblikwid program.o –o program -llikwid
```

Run:

```
likwid-perfctr -C <CPULIST> -g <GROUP> -m ./program
```

Activate markers

MPI:

```
likwid-mpirun –np 4 –pin <PINEXPR> –g <GROUP> -m ./program
```

→ One separate block of output for every marked region

So... what should I look at first?

Focus on resource utilization and instruction decomposition!

Metrics to measure:

- Operation throughput (Flops/s)
- Overall instruction throughput (IPC,CPI)
- Instruction breakdown:
  - FP instructions
  - loads and stores
  - branch instructions
  - other instructions
- Instruction breakdown to SIMD width (scalar, SSE, AVX, AVX512 for x86)
- Data volumes and bandwidths to main memory (GB and GB/s)
- Data volumes and bandwidth to different cache levels (GB and GB/s)

Useful diagnostic metrics are:

- Clock frequency (GHz)
- Power (W)

All the above metrics can be acquired using performance groups:

```
MEM_DP, MEM_SP, BRANCH, DATA, L2,  L3
```

Summary of hardware performance monitoring

- Useful only if you know what you are looking for
  - Hardware event counting bears the 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 != user work
  - Waiting time in libraries (OpenMP, MPI) may cause lots of instructions
  - → distorted application characteristic
  - Compilers may introduce instructions not visible in the source code
  - Some tools can distinguish runtime instructions from user code instructions
- Another very useful application of PM: validating performance models!
  - Roofline is data centric → measure data volume through memory hierarchy