Multicore Performance and Tools

Part 1: Topology and affinity

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, numactl, **likwid-pin***

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

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

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

LIKWID performance tools

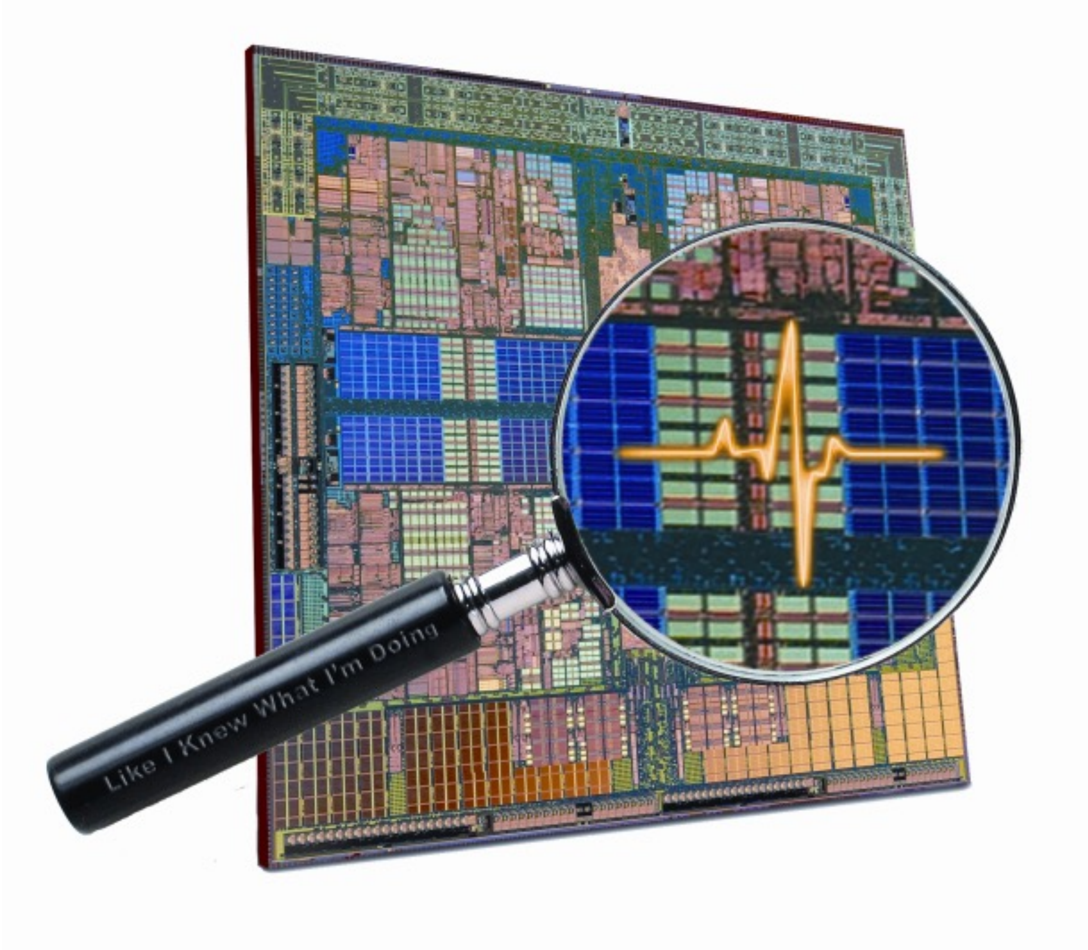
LIKWID tool suite:

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:

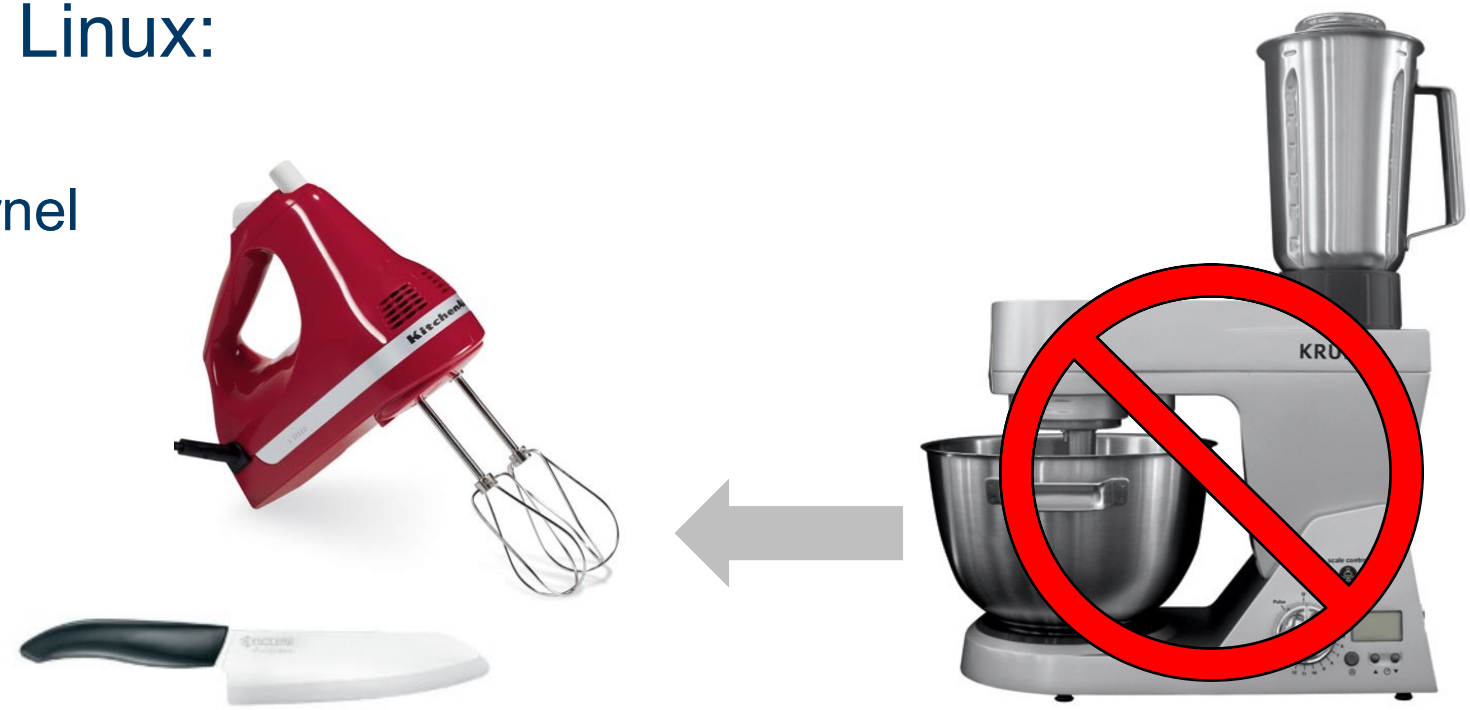
  easy to install

  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-mpirun` – Pin & measure MPI(+X) applications

`likwid-bench` - Microbenchmarking tool and environment

… some more

Reporting topology

likwid-topology

https://youtu.be/mxMWjNe73SI

Output of `likwid-topology -g`

on one node of A64FX node (OOKAMI cluster)

```
--------------------------------------------------------------------------------
CPU name:
CPU type:      Fujitsu A64FX
CPU stepping:     0
********************************************************************************
Hardware Thread Topology
********************************************************************************
Sockets:          4
Cores per socket:      12
Threads per core:      1
--------------------------------------------------------------------------------
HWThread     Thread        Core        Die        Socket        Available
0            0             0           0          0             *
1            0             1           0          0             *
…
46           0             10          0          3             *
47           0             11          0          3             *
--------------------------------------------------------------------------------
Socket 0:         ( 0 1 2 3 4 5 6 7 8 9 10 11 )
Socket 1:         ( 12 13 14 15 16 17 18 19 20 21 22 23 )
Socket 2:         ( 24 25 26 27 28 29 30 31 32 33 34 35 )
Socket 3:         ( 36 37 38 39 40 41 42 43 44 45 46 47 )
--------------------------------------------------------------------------------
********************************************************************************
Cache Topology
********************************************************************************
Level:            1
Size:             64 kB
Cache groups:     ( 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 ) ( 29 ) ( 30 ) ( 31 ) ( 32 ) ( 33 ) ( 34 ) ( 35 ) ( 36 ) ( 37 ) ( 38 ) ( 39 ) ( 40 ) ( 41 ) ( 42 ) ( 43
) ( 44 ) ( 45 ) ( 46 ) ( 47 )
--------------------------------------------------------------------------------
Level:            2
Size:             8 MB
Cache groups:     ( 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 29 30 31 32 33 34 35 ) ( 36 37 38 39 40 41 42 43 44
45 46 47 )
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
```

All physical processor IDs

Remark: System announces 4 CPU sockets but in reality its 4 CPU dies on a single socket

Output of `likwid-topology` continued

```
*******************************************************************************
NUMA Topology
*******************************************************************************
NUMA domains:      4
-------------------------------------------------------------------------------
Domain:            0
Processors:        ( 0 1 2 3 4 5 6 7 8 9 10 11 )
Distances:         10 20 30 30
Free memory:       6892.44 MB
Total memory:      8096.12 MB
-------------------------------------------------------------------------------
Domain:            1
Processors:        ( 12 13 14 15 16 17 18 19 20 21 22 23 )
Distances:         20 10 30 30
Free memory:       6733.31 MB
Total memory:      8181.69 MB
-------------------------------------------------------------------------------
Domain:            2
Processors:        ( 24 25 26 27 28 29 30 31 32 33 34 35 )
Distances:         30 30 10 20
Free memory:       7137.19 MB
Total memory:      8181.69 MB
-------------------------------------------------------------------------------
Domain:            3
Processors:        ( 36 37 38 39 40 41 42 43 44 45 46 47 )
Distances:         30 30 20 10
Free memory:       7272.06 MB
Total memory:      8161.31 MB
-------------------------------------------------------------------------------
```

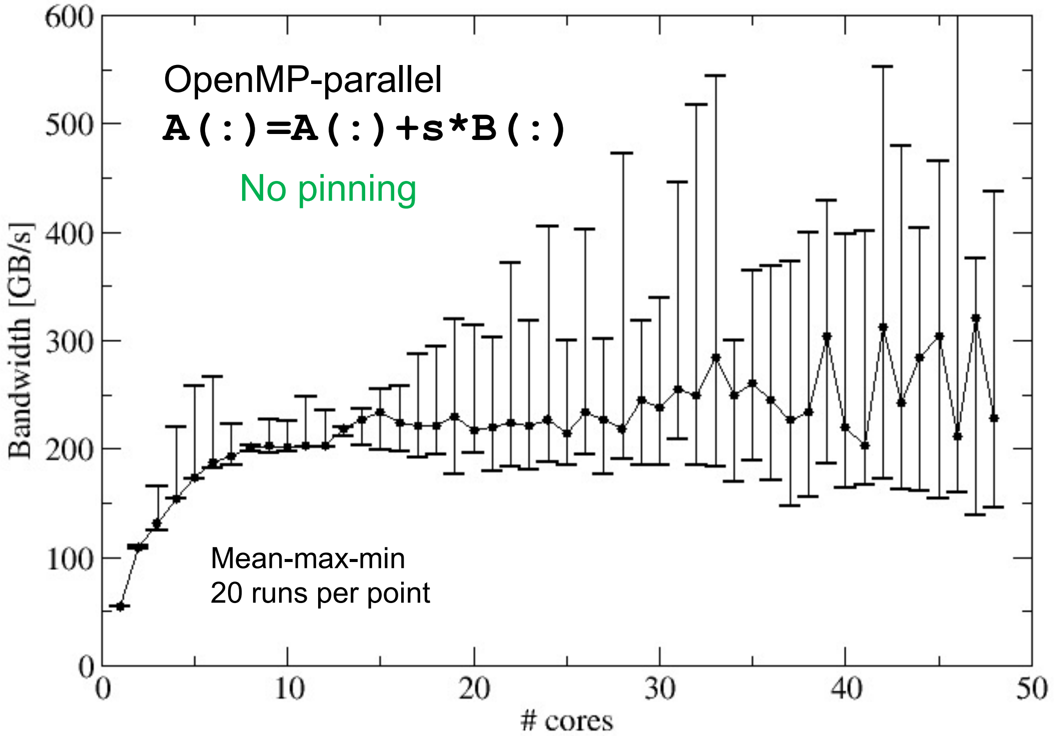
Output similar to **`numactl --hardware`**

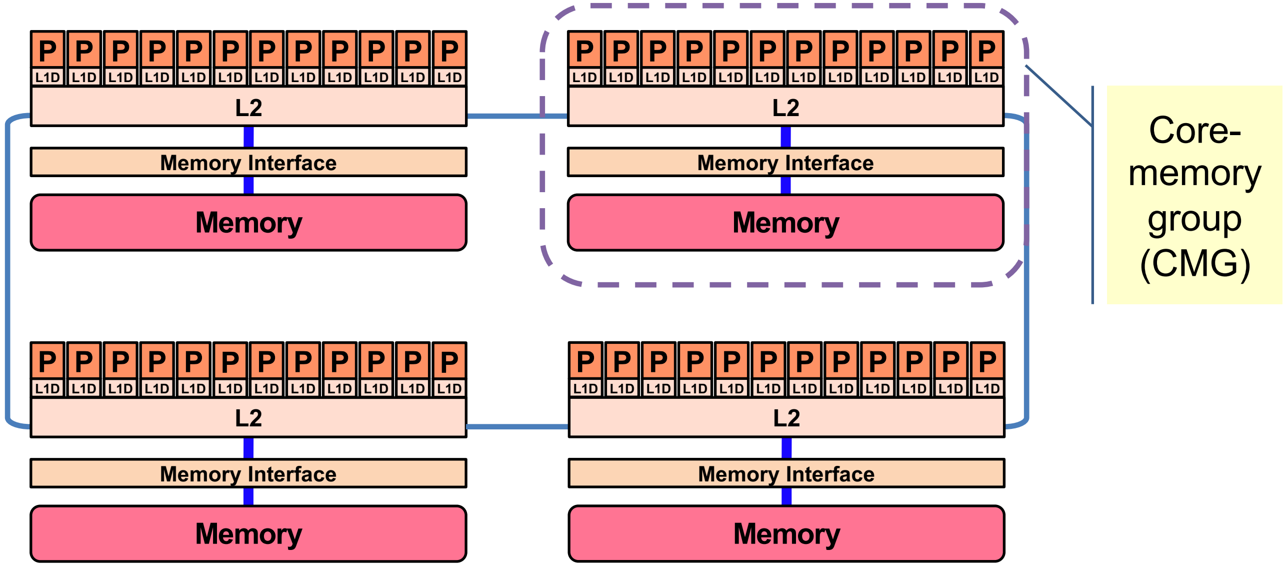
Enforcing thread/process affinity under the Linux OS

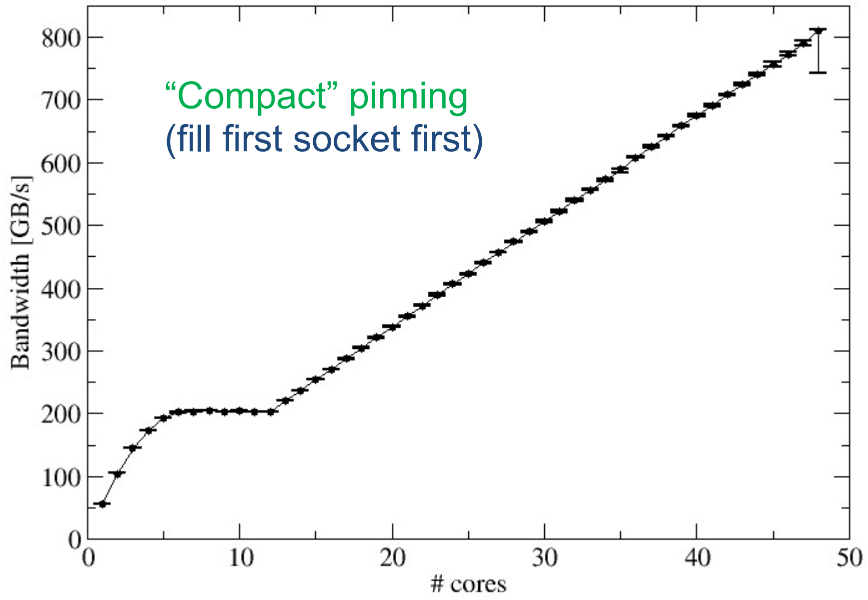
likwid-pin

https://youtu.be/PSJKNQaqwB0

DAXPY test on A64FX
Anarchy vs. thread pinning



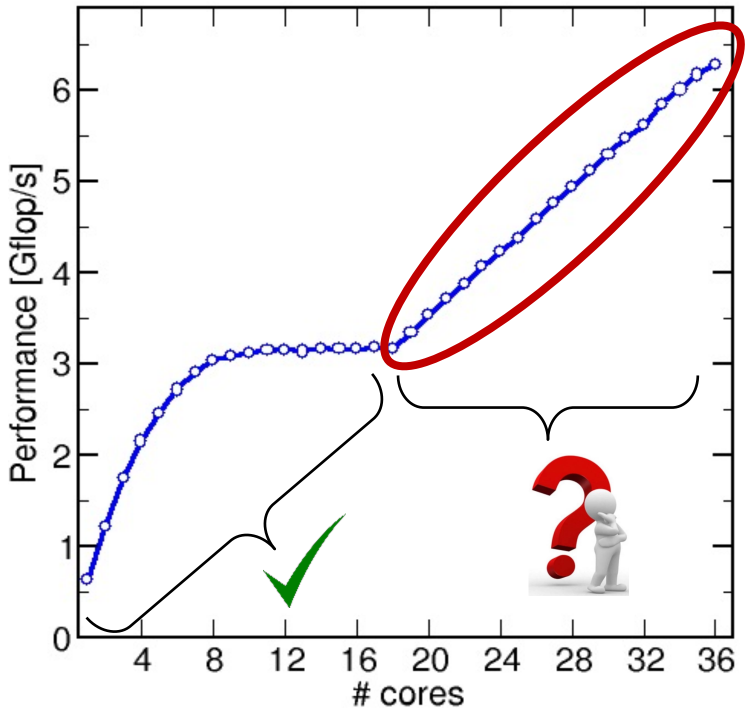




There are several reasons for caring about affinity:

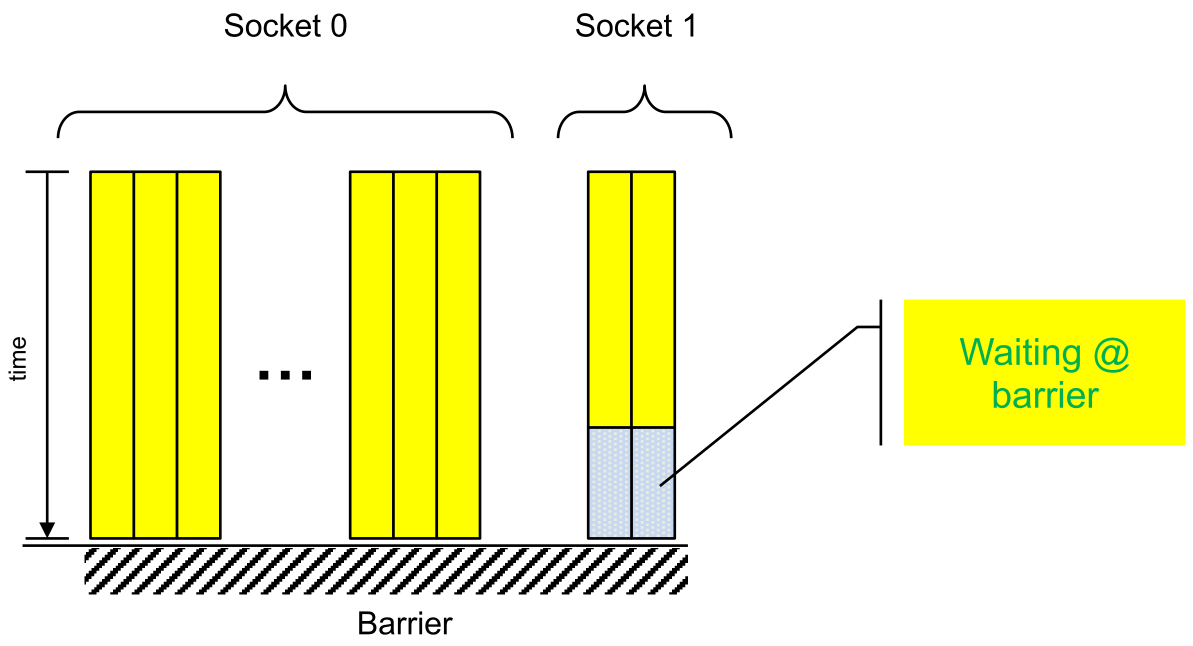
- Eliminating performance variation
- Making use of architectural features
- Avoiding resource contention

Interlude: Why the weird scaling behavior?




```
!$omp parallel do schedule(static)
 do i = 1,N
   a(i) = b(i) + s * c(i)
!$omp end parallel do
```

implicit barrier



- Every thread has the same workload
- Performance of left socket is saturated
- Barrier enforces waiting of “speeders” at sync point
- Average performance of each “right” core == average performance of each “left” core → linear scaling

More thread/process 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
  - All modern compilers with OpenMP support
  - Generic Linux: `taskset`, `numactl`, `likwid-pin` (see below)
  - OpenMP 4.0 (`OMP_PLACES`, `OMP_PROC_BIND`)
  - Slurm Batch scheduler
- 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 a node

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

Physical processors first!

```
+-------------------------------------+
| +------+ +------+ +------+ +------+ |
| |  0  4| |  1  5| | 2  6 | | 3  7 | |
| +------+ +------+ +------+ +------+ |
| +------+ +------+ +------+ +------+ |
| |  32kB| |  32kB| |  32kB| |  32kB| |
| +------+ +------+ +------+ +------+ |
| +------+ +------+ +------+ +------+ |
| | 256kB| | 256kB| | 256kB| | 256kB| |
| +------+ +------+ +------+ +------+ |
| +---------------------------------+ |
| |               8MB               | |
| +---------------------------------+ |
+-------------------------------------+
```

LIKWID Currently available thread domains

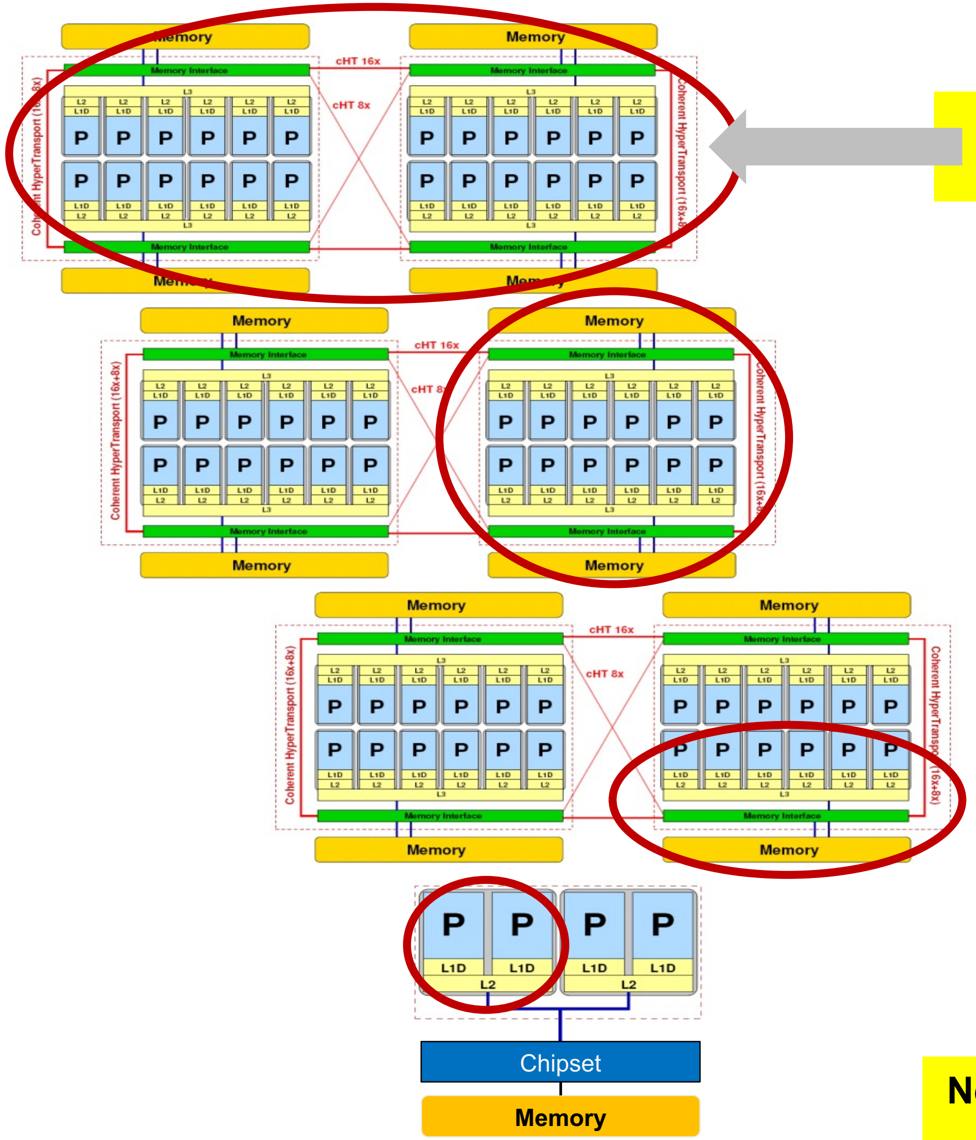
Possible unit prefixes

N node

S socket

M NUMA domain

C outer level cache group



Default if –c is not specified!

New domain Dx for CPU die in upcoming version

Advanced options for pinning: Expressions

- Expressions are more powerful in situations where the pin mask would be very long or clumsy

  Compact pinning (counting through HW threads):

  ```
  $ likwid-pin -c E:<thread domain>:\
      <number of threads>\
      [:<chunk size>:<stride>] ...
  ```

  Scattered pinning across all domains of the designated type:

  ```
  $ likwid-pin -c <domaintype>:scatter
  ```

- Examples:

  ```
  $ likwid-pin -c E:N:8:1:2 ...
  $ likwid-pin -c E:N:120:2:4 ...
  ```

- Scatter across all NUMA domains:

  ```
  $ likwid-pin -c M:scatter
  ```

“Compact” placement!

```
+-------------------------------------+
| +------+ +------+ +------+ +------+ |
| |  0  1| |  2  3| | 4  5 | | 6  7 | |
| +------+ +------+ +------+ +------+ |
| +------+ +------+ +------+ +------+ |
| |  32kB| |  32kB| |  32kB| |  32kB| |
| +------+ +------+ +------+ +------+ |
| +------+ +------+ +------+ +------+ |
| | 256kB| | 256kB| | 256kB| | 256kB| |
| +------+ +------+ +------+ +------+ |
| +---------------------------------+ |
| |               8MB               | |
| +---------------------------------+ |
+-------------------------------------+
```

Example: **`likwid-pin`** with Intel OpenMP

Running the STREAM benchmark with **`likwid-pin`**:

```
$ likwid-pin -c S0:0-3 ./stream
-------------------------------------------------------------
STREAM version $Revision: 5.10 $
-------------------------------------------------------------
This system uses 8 bytes per array element.
-------------------------------------------------------------
Array size = 100000000 (elements), Offset = 0 (elements)
Memory per array = 762.9 MiB (= 0.7 GiB).
Total memory required = 2288.8 MiB (= 2.2 GiB).
Each kernel will be executed 10 times.
-------------------------------------------------------------
[pthread wrapper]
[pthread wrapper] MAIN -> 0
[pthread wrapper] PIN_MASK: 0->1  1->2  2->3
[pthread wrapper] SKIP MASK: 0x0
        threadid 281473873604960 -> hwthread 1 - OK
        threadid 281473865150816 -> hwthread 2 - OK
        threadid 281473856696672 -> hwthread 3 - OK
Number of Threads requested = 4
Number of Threads counted = 4
  [... 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

A64FX with 48 cores, 1x12 cores, 1 thread per physical core, fill 1 CMG

```
OMP_NUM_THREADS=12
OMP_PLACES=cores
OMP_PROC_BIND=close
```

Always prefer abstract places instead of HW thread IDs!

A64FX with 48 cores, 24 cores to be used, 2 threads per physical core

```
OMP_NUM_THREADS=24
OMP_PLACES=cores(12)
OMP_PROC_BIND=close    # spread will also do
```