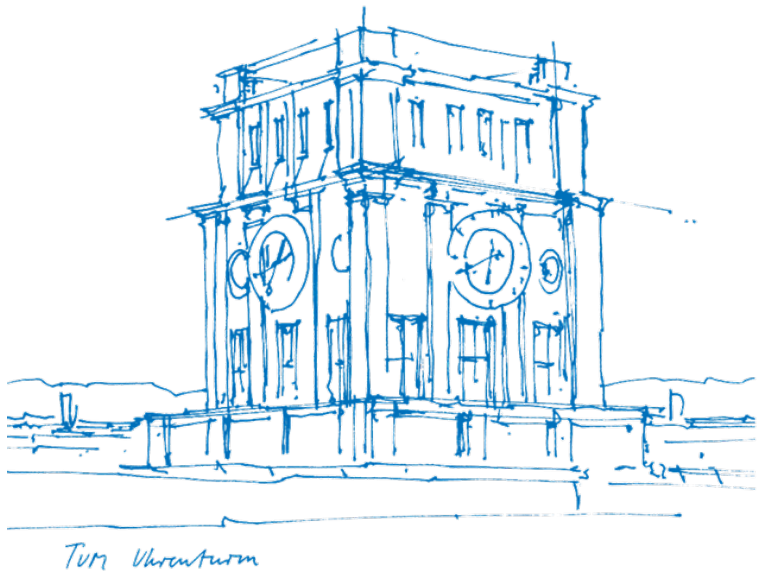
HPC Lab

Session 2: OpenMP

Carsten Uphoff, Chaulio Ferreira, Michael Bader
TUM – SCCS

6 November 2017



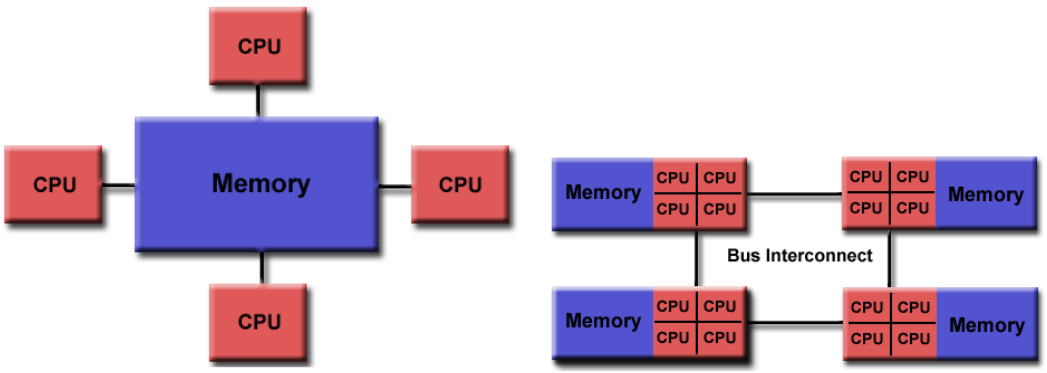
What is OpenMP?

“OpenMP is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify high-level parallelism in Fortran and C/C++ programs”

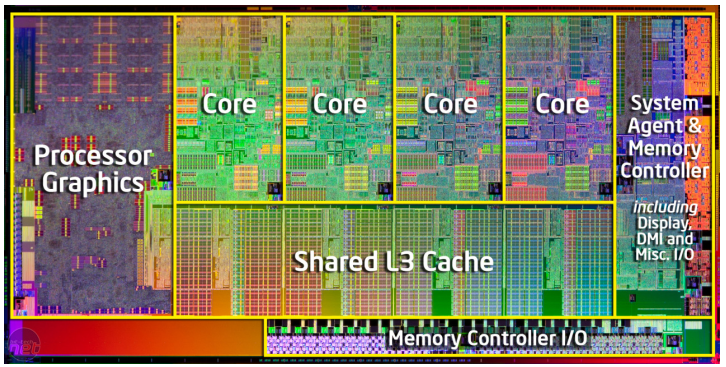
`http://openmp.org/openmp-faq.html#WhatIs`

Shared-Memory-Parallelism

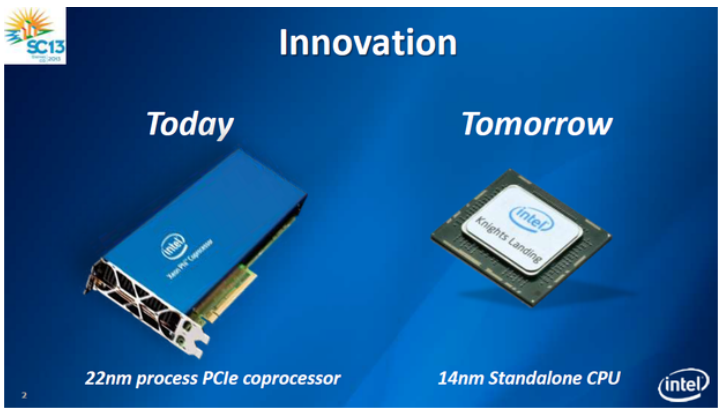
- Transparent programming view: Cores access “same” memory
- In hardware: Be aware of layout, e.g. NUMA



https://computing.llnl.gov/tutorials/openMP/

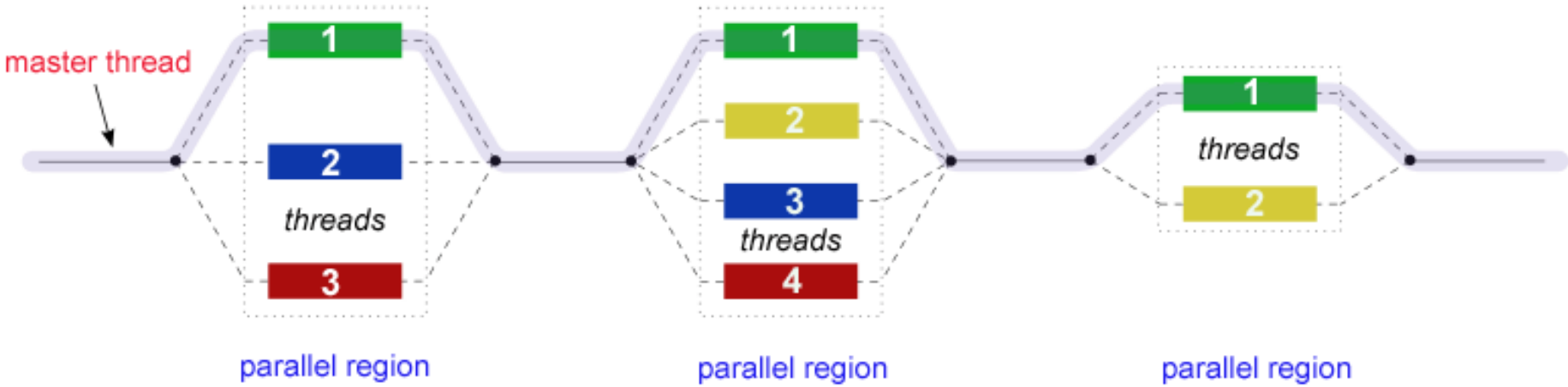


Sandy Bridge CPU, http://www.bit-tech.net/hardware/cpus/2011/01/03/intel-sandy-bridge-review/1



Knights Corner and Knights Landing, http://www.zdnet.com/sc13-intel-reveals-knights-landing-high-performance-cpu-7000023393/

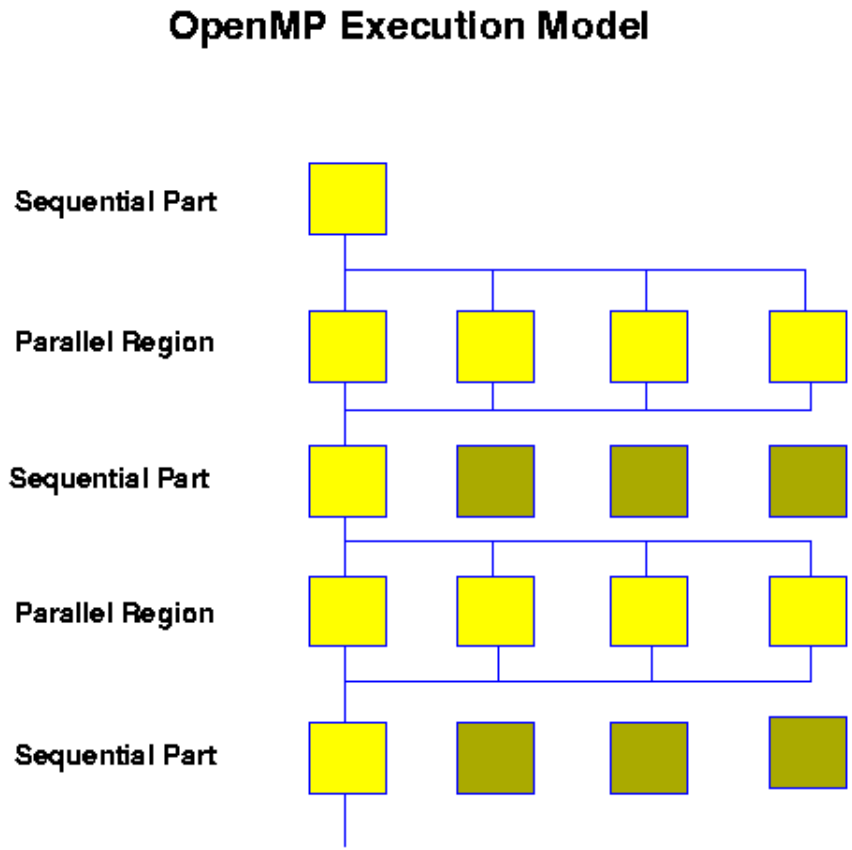
Fork-Join Model



https://computing.llnl.gov/tutorials/openMP/#ProgrammingModel

A Simple Loop

```
1 #include <omp.h>
2 // some initialization
3
4 #pragma omp parallel for
5 for (i = 0; i < n; i++) {
6   for (j = 0; j < n; j++) {
7     // do some work
8   }
9 }
```



OpenMP - Parallel programming on shared memory systems
https://www.lrz.de/services/software/parallel/openmp/

OMP: An Overview

- Compiler-directives:
  `#pragma omp ...[clause[[,] clause]...]`
  - Parallel region
    `#pragma omp parallel`
  - Work-sharing
    `#pragma omp for ..., and many more`
  - Synchronization
    `#pragma omp barrier, and many more`
- Data Scope Clauses
  `shared, private, firstprivate, reduction, ...`
- Library routines
  `omp_get_num_threads(), omp_get_thread_num(), ...`

Compilation and Execution

- compile with additional Compiler-Flag `-qopenmp`:
  `icpc -qopenmp -o my_algorithm alg.c`
- `export OMP_NUM_THREADS=number_of_threads`
  `./my_algorithm`

Parallel Region

```
#pragma omp parallel [clause[[,] clause]...]
structured block
```

- code within parallel region is executed by all threads

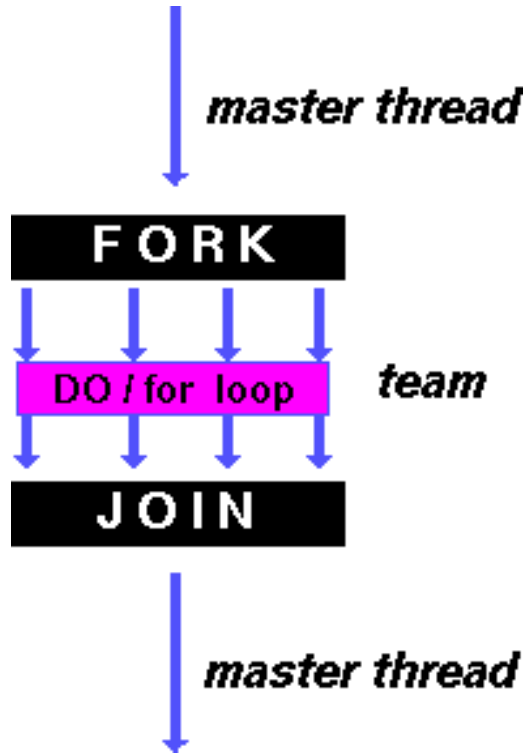
```
#include <omp.h>
// more initialization ...

#pragma omp parallel
{
  int size = omp_get_num_threads();
  int rank = omp_get_thread_num();
  printf("Hello World! (Thread %d of %d)", rank, size);
}
```

Work Sharing: `for`

```
#pragma omp for [clause[[,] clause]...]
for-loop
```

- within a *parallel* region and directly in front of a *for*-loop
- iterations are scheduled across different threads
- implicit synchronization at the end of *for*-loop (can be disabled with *nowait* clause)
- shortcut possible: `#pragma omp parallel for`

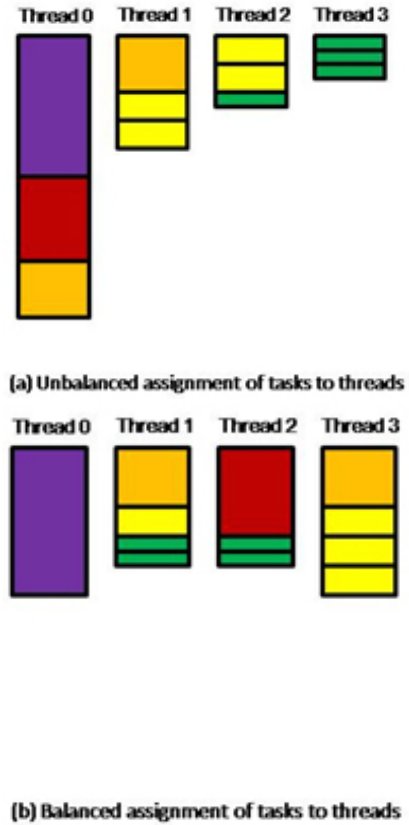


https://computing.llnl.gov/tutorials/openMP/

Work Sharing: `for`

```
#pragma omp for [clause[[,] clause]...]
for-loop
```

- *schedule* clause determines how to map iterations to threads
  - *schedule(static[, chunk size])* default chunk size is #iterations divided by #threads
  - *schedule(dynamic[,chunk size])* default chunk size is 1
  - *schedule(guided[,chunk size])* similar to dynamic but initial chunk size as in static
  - *schedule(runtime)* Scheduling is given by environment variable OMP_SCHEDULE



Intel, Load Balance and Parallel Performance, `https://software.intel.com/en-us/articles/load-balance-and-parallel-performance`

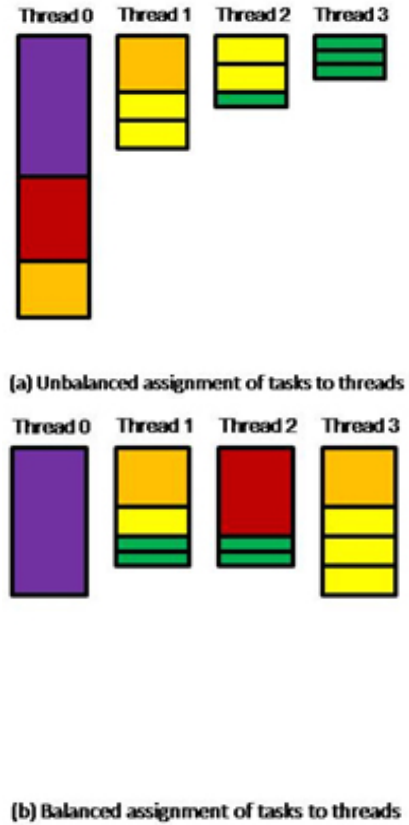
Work Sharing: `task`

```
#pragma omp task [clause[[,] clause]...]
structured block
```

where `clause` can be...

- `final(expression)`: if *expression* is *true*, task is executed sequentially; no recursive task generation
- `untied`: the execution of a task is not tied to one single thread
- `shared | firstprivate | private [...]`

`#pragma omp taskwait`: waits for children completion



Intel, Load Balance and Parallel Performance, `https://software.intel.com/en-us/articles/load-balance-and-parallel-performance`

“Work Sharing”: `single`

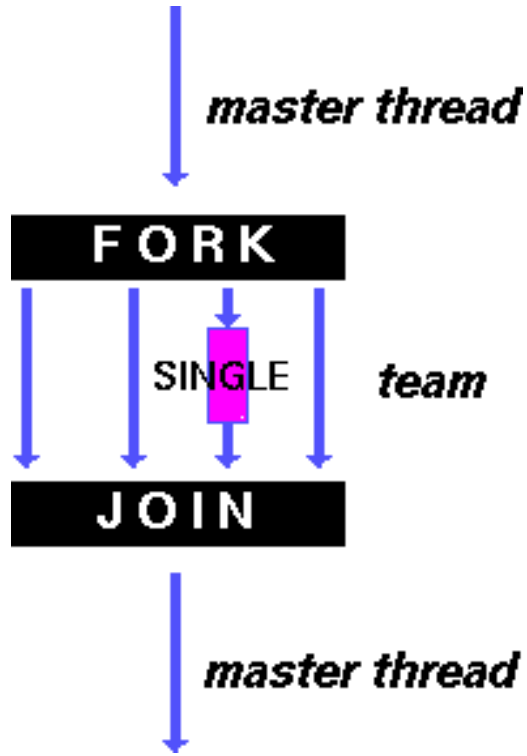
```
#pragma omp single [clause[[,] clause]...]
structured block
```

where `clause` can be...

- use only one (arbitrary) thread
- implicit synchronization (can be disabled with `nowait` clause)

```
#pragma omp master
structured block
```

- use only master thread for structured block
- NO synchronization at the end



`https://computing.llnl.gov/tutorials/openMP/`

Synchronization

- `#pragma omp barrier`: block execution until all threads have reached the barrier
- `#pragma omp critical`:
  `structured block`
  - only one thread at a time can execute the structured block encapsulated by critical
  - *Warning:* Use carefully; can definitely kill performance.

Reduction

`reduction(operator: list)`

- executes a reduction of variables in list using operator
- available operators: `+`, `*`, `-`, `&&`, `||`
  since OpenMP 3.1: `min`, `max`

```
1 #pragma omp parallel for private(r), reduction(+: sum)
2 for(i = 0; i < n; i++) {
3   r = compute_r(i);
4   sum = sum + r;
5 }
```

Scopes

- `private(list)`: *declares* variables in list as private for each thread *(no copy)*
- `shared(list)`: variables in list are used by all threads (race conditions are possible), write accesses have to be handled by the programmer
- `firstprivate(list)`: private variables *and* init them with the latest valid value before parallel region
- `lastprivate(list)`: variable in the serial part receives the value from the thread executing the last parallel iteration
- ...
- default data scope is shared, but exceptions exist $\Rightarrow$ be precise
  - local variables are always *private*
  - loop-variables of *for*-loops are private

Fill the Gaps

```
1 k = compute_k();
2
3 #pragma omp parallel for private(?), shared(?), firstprivate(?), lastprivate(?),
       reduction(?)
4 for (i = 0; i < n; i++) {
5   k = compute_my_k(i, k);
6   r = compute_r(i, k, h);
7   sum = sum + r;
8 }
```

Nested parallelism

Enable with `OMP_NESTED` or `omp_set_nested(1)`.

```
1  omp_set_nested(1);
2
3  #pragma omp parallel num_threads(4)
4  {
5    printf("outer thread id %d\n", omp_get_thread_num());
6  #pragma omp parallel num_threads(2)
7    {
8      printf("inner thread id %d\n", omp_get_thread_num());
9    }
10 }
```

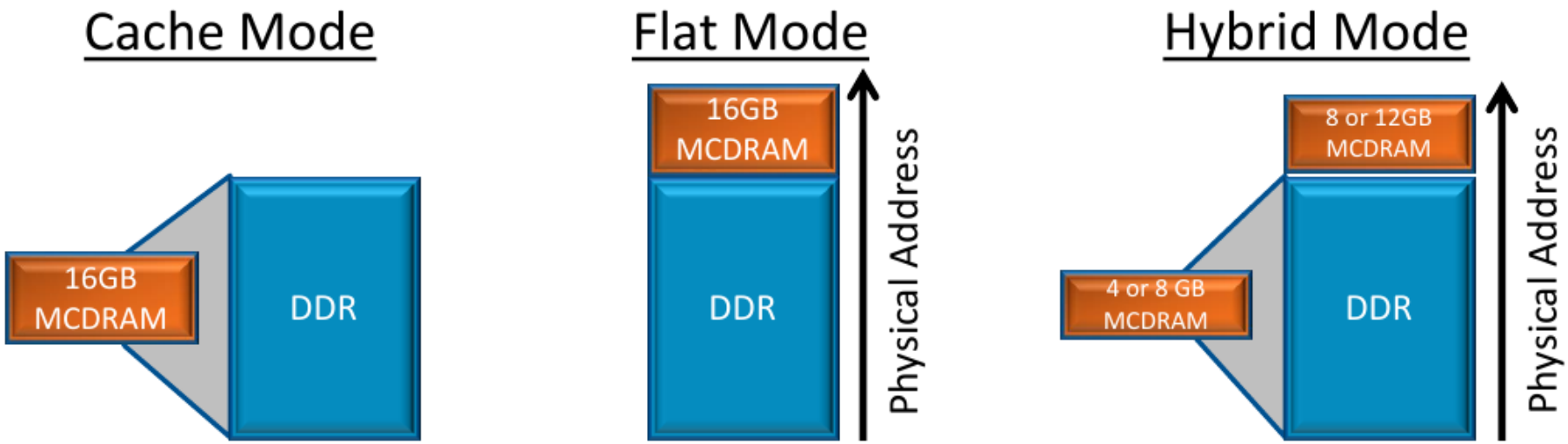
- Not nested: 4 threads
- Nested: 8 threads

KNL memory modes

Two kinds of memory.

- DDR: up to 384 GB, off-chip, slow
- MCDRAM: 16 GB, on-chip, fast

Memory mode selected at boot time:



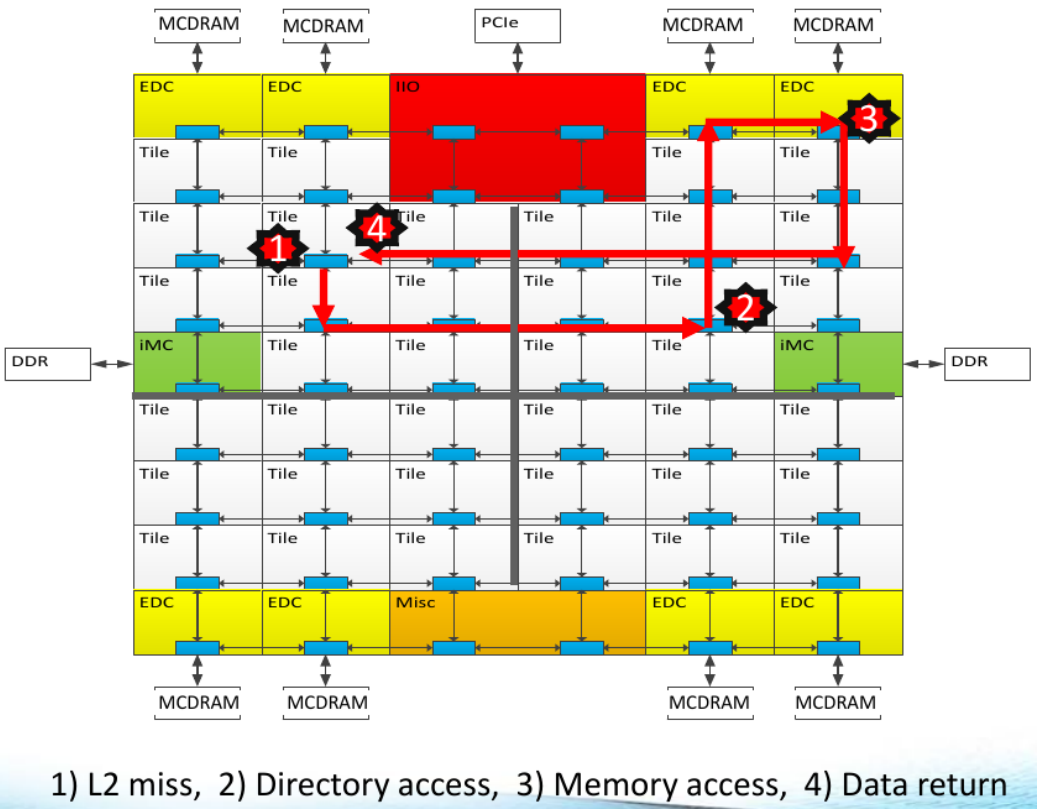
https://www.alcf.anl.gov/files/HC27.25.710-Knights-Landing-Sodani-Intel.pdf

KNL cluster modes

Three agents:

1. Core – Initiates a memory request
2. Tag directory – Memory request is forwarded to tag directory which knows the dedicated memory channel
3. Memory channel – Memory channel sends request to memory

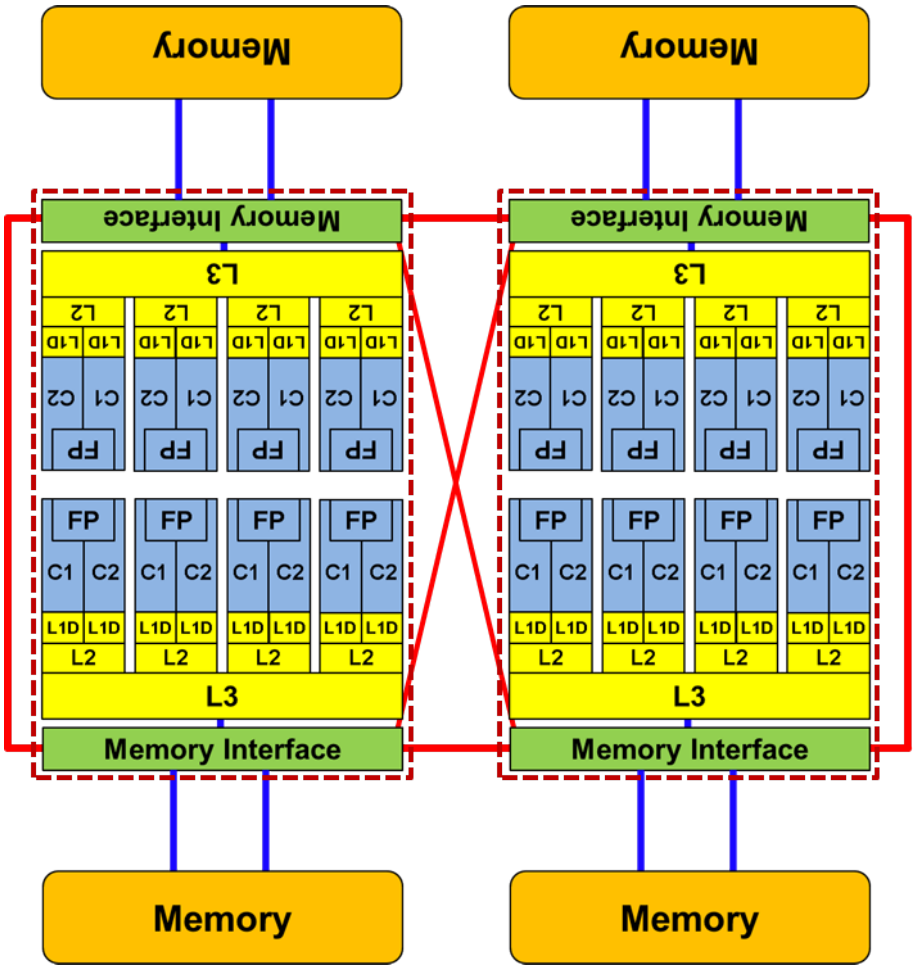
https://www.alcf.anl.gov/files/HC27.25.710-Knights-Landing-Sodani-Intel.pdf



Cluster mode (All2All, Quadrant, SNC-2/4) determines affinity of tag directory and memory channel. SNC exposes quadrants as NUMA domains.

NUMA

- Non-uniform memory access (NUMA): Not all cores access all memory in the same manner
- Pinning: Use close-to-core memory as far as possible
  - Linux first touch policy: *not the malloc counts but the first (write) access*
  - `numactl` is a useful tool



Georg Hager's Blog,
`http://blogs.fau.de/hager/`