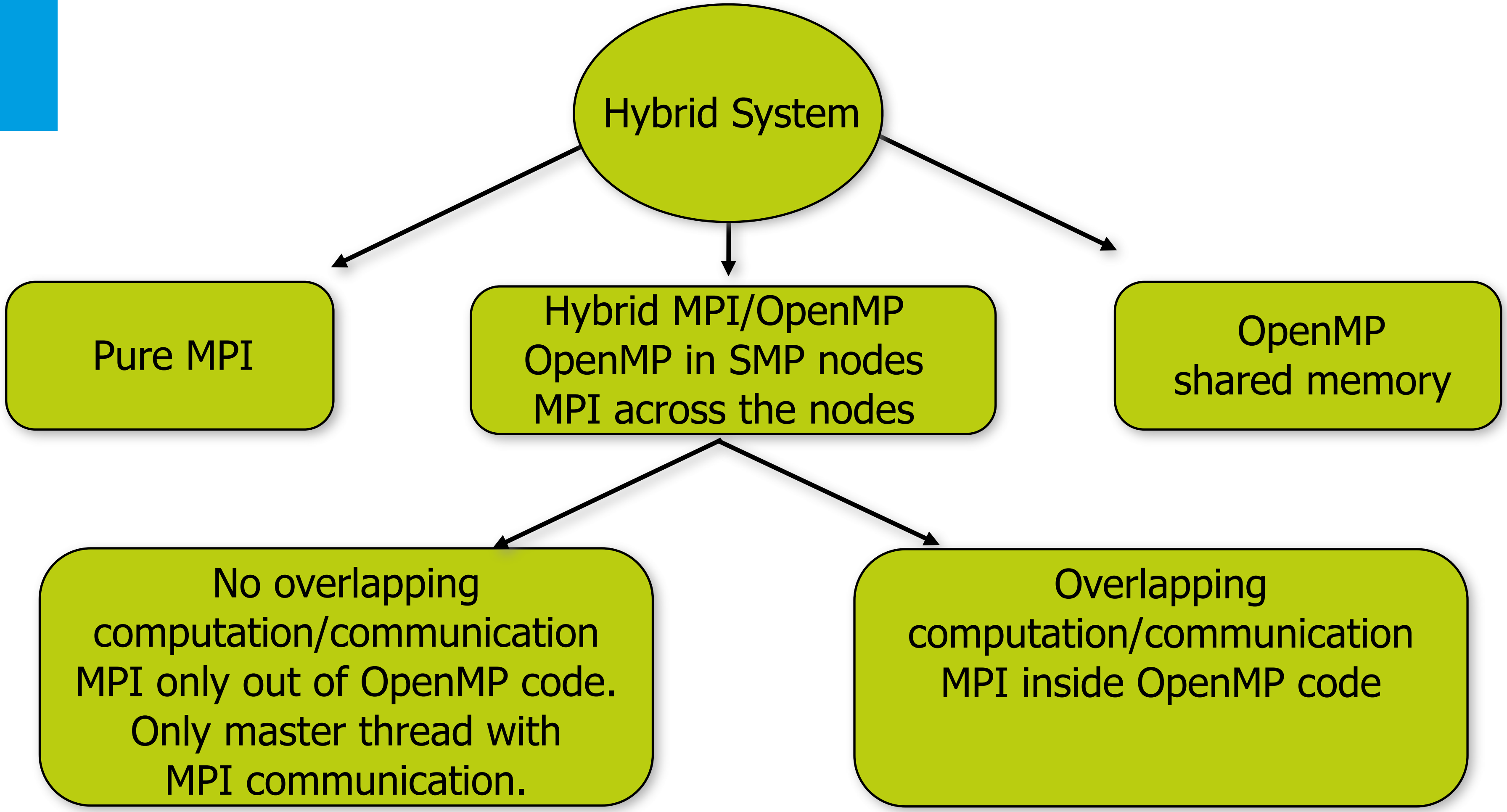
Programming with MPI

Hybrid MPI + OpenMP

Jan Thorbecke

Hybrid systems programming hierarchy



What Is OpenMP?

- Compiler directives for multithreaded programming
- Easy to create threaded Fortran and C/C++ codes
- Supports data parallelism model
- Portable and Standard
- Incremental parallelism
  - ➡Combines serial and parallel code in single source

Directive based

- Directives are special comments in the language

– Fortran fixed form: `!$OMP, C$OMP, *$OMP`

– Fortran free form: `!$OMP`

Special comments are interpreted by OpenMP compilers

```
      w = 1.0/n
      sum = 0.0
!$OMP PARALLEL DO PRIVATE(x) REDUCTION(+:sum)
      do I=1,n
        x = w*(I-0.5)
        sum = sum + f(x)
      end do
      pi = w*sum
      print *,pi
      end
```

Comment in Fortran
but interpreted by
OpenMP compilers

C example

- `#pragma omp` directives in C
  - Ignored by non-OpenMP compilers

```
  w = 1.0/n;
  sum = 0.0;
#pragma omp parallel for private(x) reduction(+:sum)
  for(i=0, i<n, i++) {
    x = w*((double)i+0.5);
    sum += f(x);
  }
  pi = w*sum;
  printf("pi=%g\n", pi);
}
```

Data Environment

- OpenMP uses a shared-memory programming model
    - Most variables are shared by default.
    - Global variables are shared among threads
      C/C++: File scope variables, static
- Not everything is shared, there is often a need for “local” data as well

About Variables in SMP

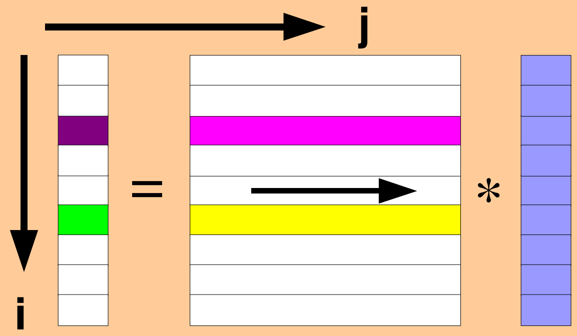
- Shared variables
  Can be accessed by every thread thread. Independent read/write operations can take place.

- Private variables
  Every thread has it's own copy of the variables that are created/ destroyed upon entering/leaving the procedure. They are not visible to other threads.

serial code	**parallel code**
global	shared
auto local	local
static	use with care
dynamic	use with care

Matrix-vector example

```
#pragma omp parallel for default(none) \
            private(i,j,sum) shared(m,n,a,b,c)
for (i=0; i<m; i++)
{
   sum = 0.0;
   for (j=0; j<n; j++)
     sum += b[i][j]*c[j];
   a[i] = sum;
 }
```



TID = 0

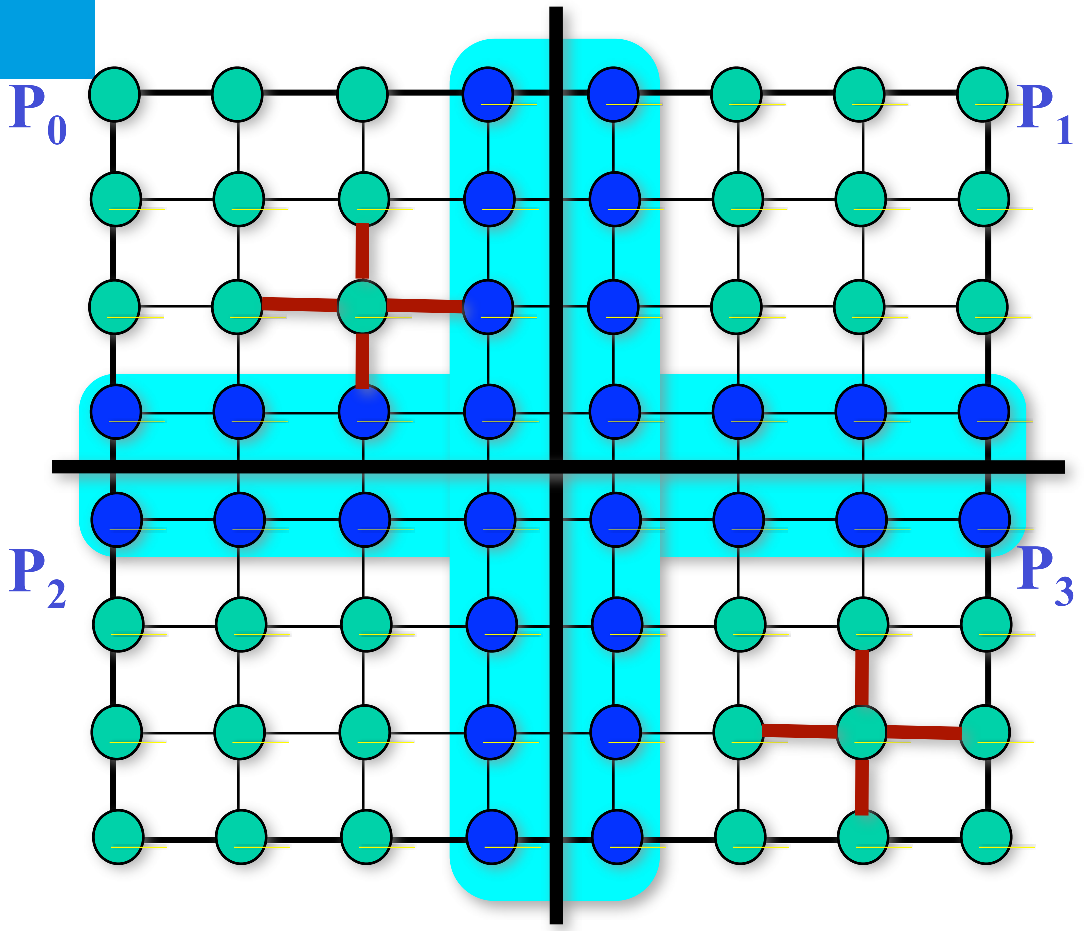
```
for (i=0,1,2,3,4)
 i = 0
sum = Σ  b[i=0][j]*c[j]
   a[0] = sum
 i = 1
sum = Σ  b[i=1][j]*c[j]
   a[1] = sum
```

TID = 1

```
for (i=5,6,7,8,9)
 i = 5
sum = Σ  b[i=5][j]*c[j]
   a[5] = sum
 i = 6
sum = Σ  b[i=6][j]*c[j]
   a[6] = sum
```

etc

Overlapping computation/communication: Example



Suppose we wish to solve the PDE

$$-\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) = f(x,y)$$

Using the Jacobi method: the value of u at each discretization point is given by a certain average among its neighbors, until convergence.

Distributing the mesh to SMP clusters by Domain Decomposition, it is clear that the GREEN nodes can proceed without any comm., while the Blue nodes have to communicate first and calculate later.

MPI/OpenMPI: Overlapping computation/ communication

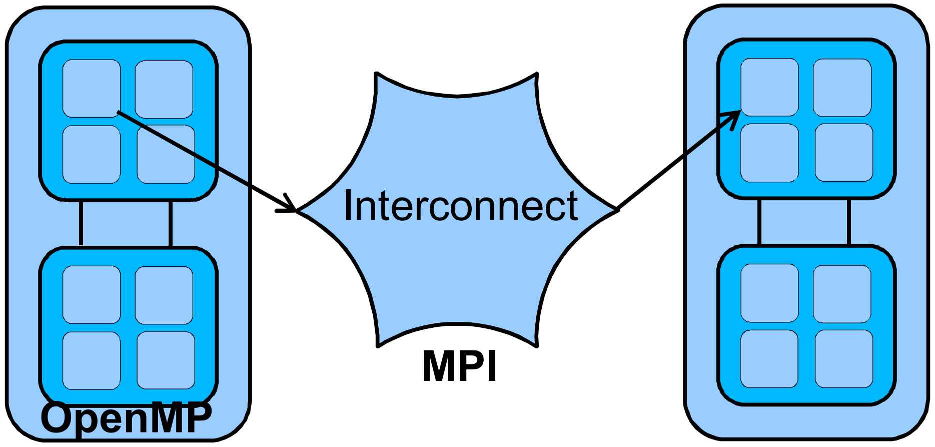
Not only the master but other threads communicate. Call MPI primitives in OpenMP code regions.

```
if (my_thread_id < # ){
    MPI_… (communicate needed data)
} else
    /* Perform computations that to not need
    communication */
    .
    .
}
/* All threads execute code that requires
    communication */
    .
    .
```

```
for (k=0; k < MAXITER; k++){
    /* Start parallel region here */
    #pragma omp parallel private(){
            my_id = omp_get_thread_num();

            if (my_id is given “halo points”)
                    MPI_SendRecv(“From neighboring MPI process”);
            else{
                    for (i=0; i < # allocated points; i++)
                            newval[i] = avg(oldval[i]);
            }

            if (there are still points I need to do) /* Thi
                    for (i=0; i< # remaining points; i++)
                            newval[i] = avg(oldval[i]);

            }
            for (i=0; i<(all_my_points); i++)
                            oldval[i] = newval[i];
    }
    MPI_Barrier(); /* Synchronize all MPI processes here */
}
```

Hybrid programming

- Parallel programming model combining:
  - Parallelization over one SMP node with shared-memory parallelization
  - Parallelization over parallel computer with message passing
- Here: MPI + OpenMP



Matrix vector OpenMP

```
#pragma omp parallel for \
  shared(A,x,n) private(i,j) \
  reduction (+:y) \
  schedule(guided,chunk)
  for (i=0;i<n;i++){
   for (j=0;j<n;j++)
      y[i] += A[i][j]*x[j];
 }
```

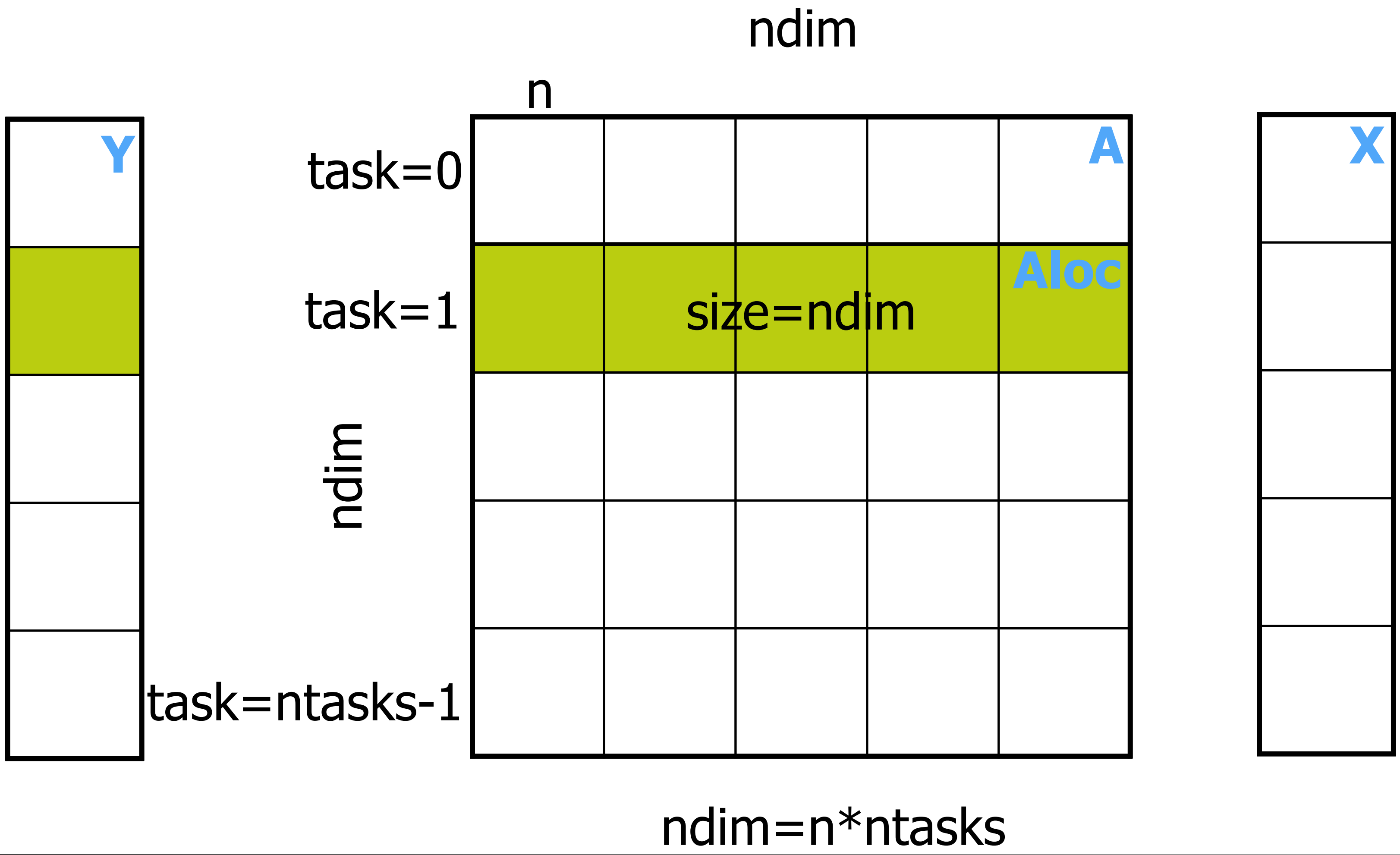
```
#pragma omp parallel for \
 shared(A,y,x,n) private(i,j,asum) \
  schedule(guided,chunk)
  for (i=0;i<n;i++){
    asum=0.0;
    for (j=0;j<n;j++)
      asum += A[i][j]*x[j];
    y[i]=asum;
  }
```

Exercise: MatrixVector part 2

- From directory MatrixVector
  - Use MPI calls from previous exercise (solution: mvx_mpi.c / f90)
  - Insert OpenMP directives
  - insert OpenMP directives for local loop (see previous slide)
    - compile with: `mpicc -fopenmp mxv_mpi_hyb.c`
  - contains 2 OpenMP based solutions
  - check performance running pure MPI and Hybrid, use job.slurm to submit job to queue

  - solution in mxv_mpi_hyb.c / f90

Collectives: Matrix-Vector



Thread support in MPI

- MPI standard defines four levels of support
  - MPI_THREAD_SINGLE
    - Only one thread allowed
  - MPI_THREAD_FUNNELED
    - Only master thread allowed to make an MPI call
  - MPI_THREAD_SERIALIZED
    - All threads allowed to make MPI calls, but not concurrently
  - MPI_THREAD_MULTIPLE
    - No restrictions

Affinity

- Binding of MPI ranks and OpenMP threads to resources, core, hypertherads
- TODO srun examples from UWCW…

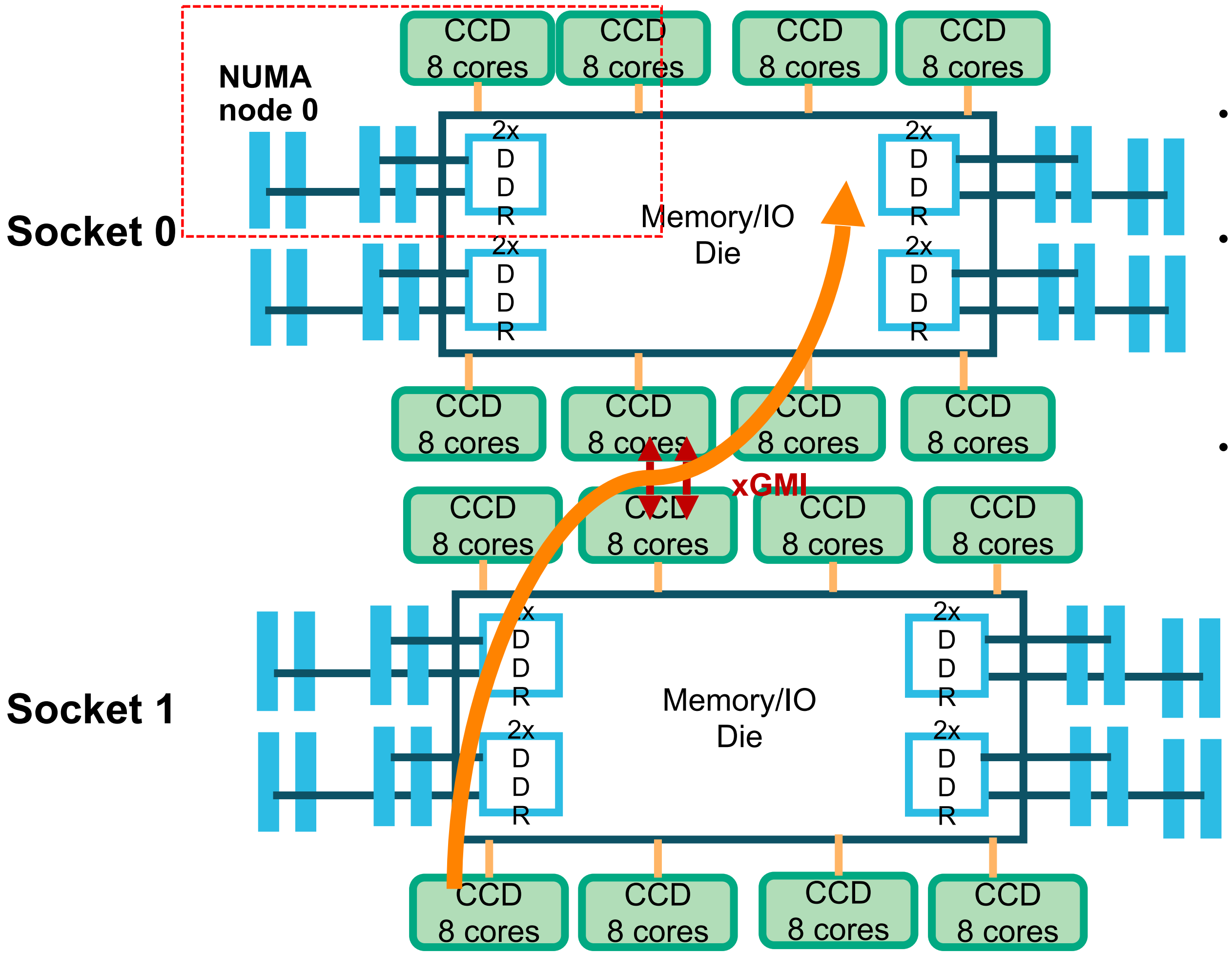
A quick recap – glossary of terms

- **Hardware**
  - **Socket**
    The hardware you can touch and insert into the mother board
  - **CPU**
    The minimum piece of hardware capable of running a Software Task. It may share some or all its hardware resources with other CPUs
    Equivalent to a single "Intel Hyperthread" or AMD SMT Thread.
  - **Core**
    The individual unit of hardware for processing, part of the CPU. This can be called a compute unit (CU)
  - This terminology is used to cover hardware from multiple vendors

- **Software**
  - **Task**
    A discrete software process with an individual address space. One task is equivalent to a UNIX process, MPI Rank, Coarray Image, UPC Thread, or SHMEM PE. This can also be called a Processing Element (PE)
  - **Threads**
    A logically separate stream of execution inside a parent Task that shares the same address space (OpenMP, Pthreads)
  - Different software approaches also use different naming convention. This is the software-neutral convention we are going to use

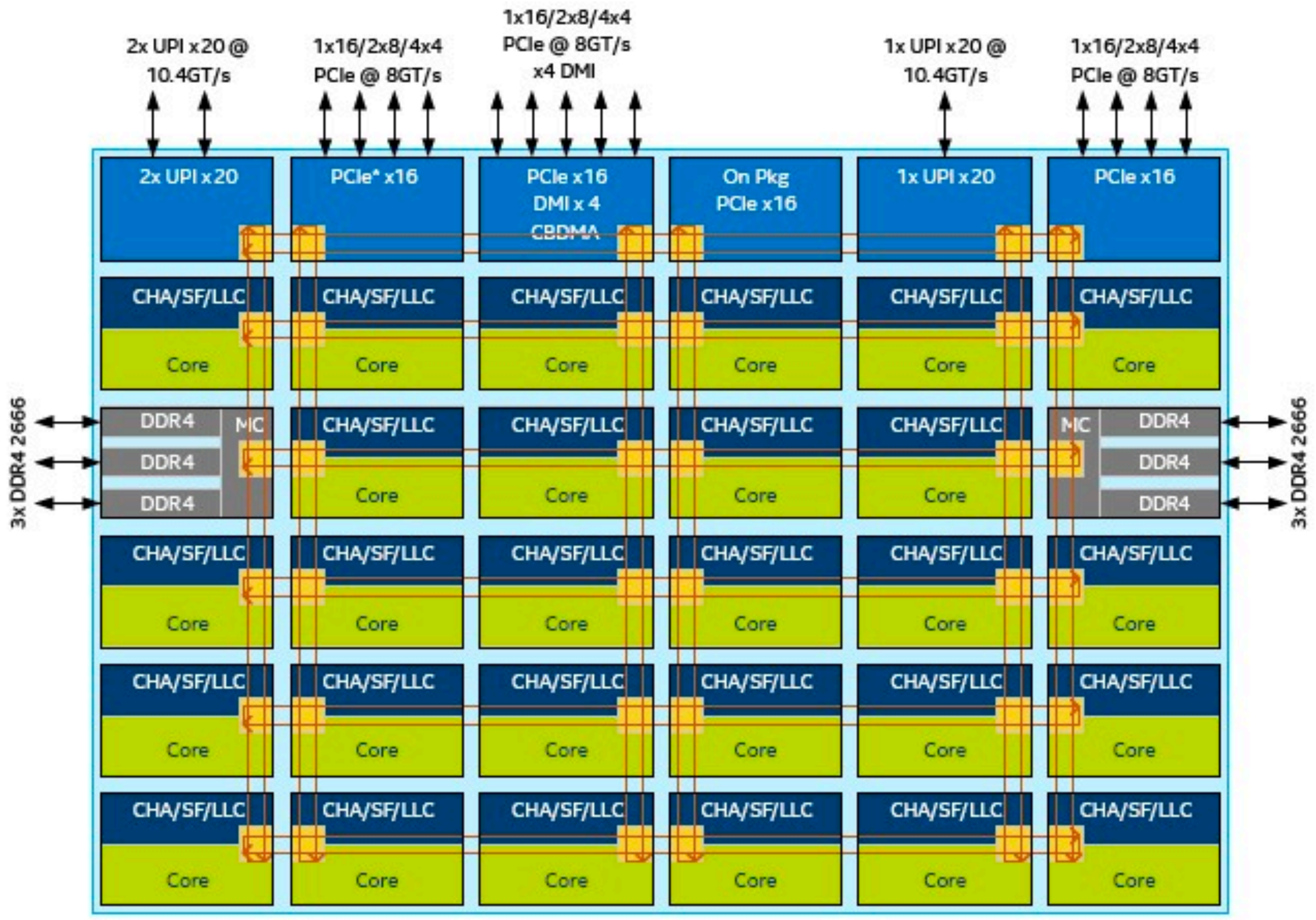
- **The concept of mapping tasks or threads to hardware is crucial for optimal performance.**

Numa nodes AMD Milan



- **Each node is divided into eight NUMA nodes, associated with the two sockets/dies.**
- **The design of the node means that CPUs accessing data stored on the other socket/die must cross the xGMI inter-processor bus.**
- **This is marginally slower than accessing local memory and creates "Non-Uniform Memory Access" (NUMA) regions.**

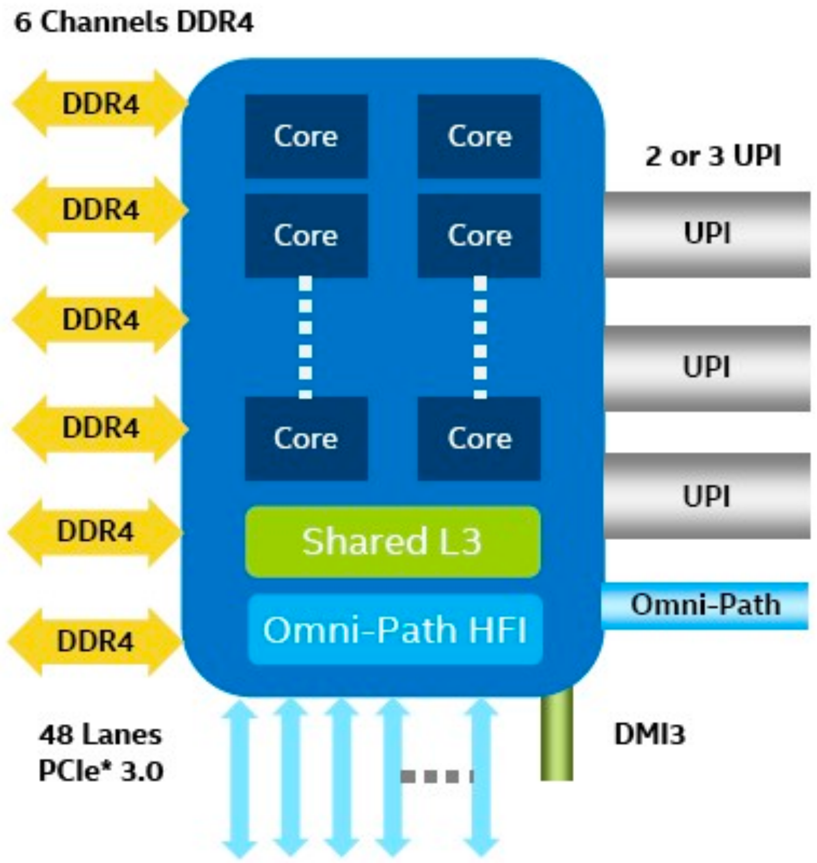
NUMA nodes Intel Skylake



CHA – Caching and Home Agent ; SF – Snoop Filter; LLC – Last Level Cache;
Core – Skylake-SP Core; UPI – Intel® UltraPath Interconnect

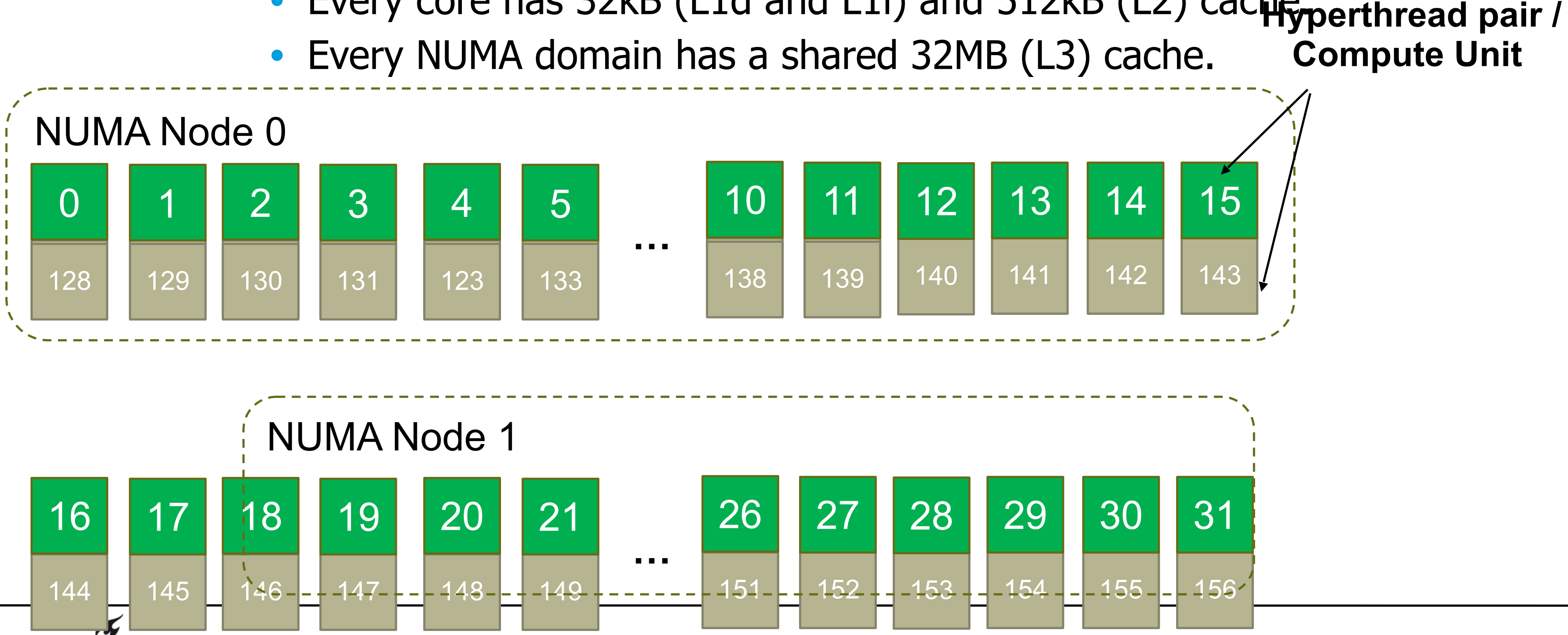
TU Delft

NUMA nodes Intel Skylake



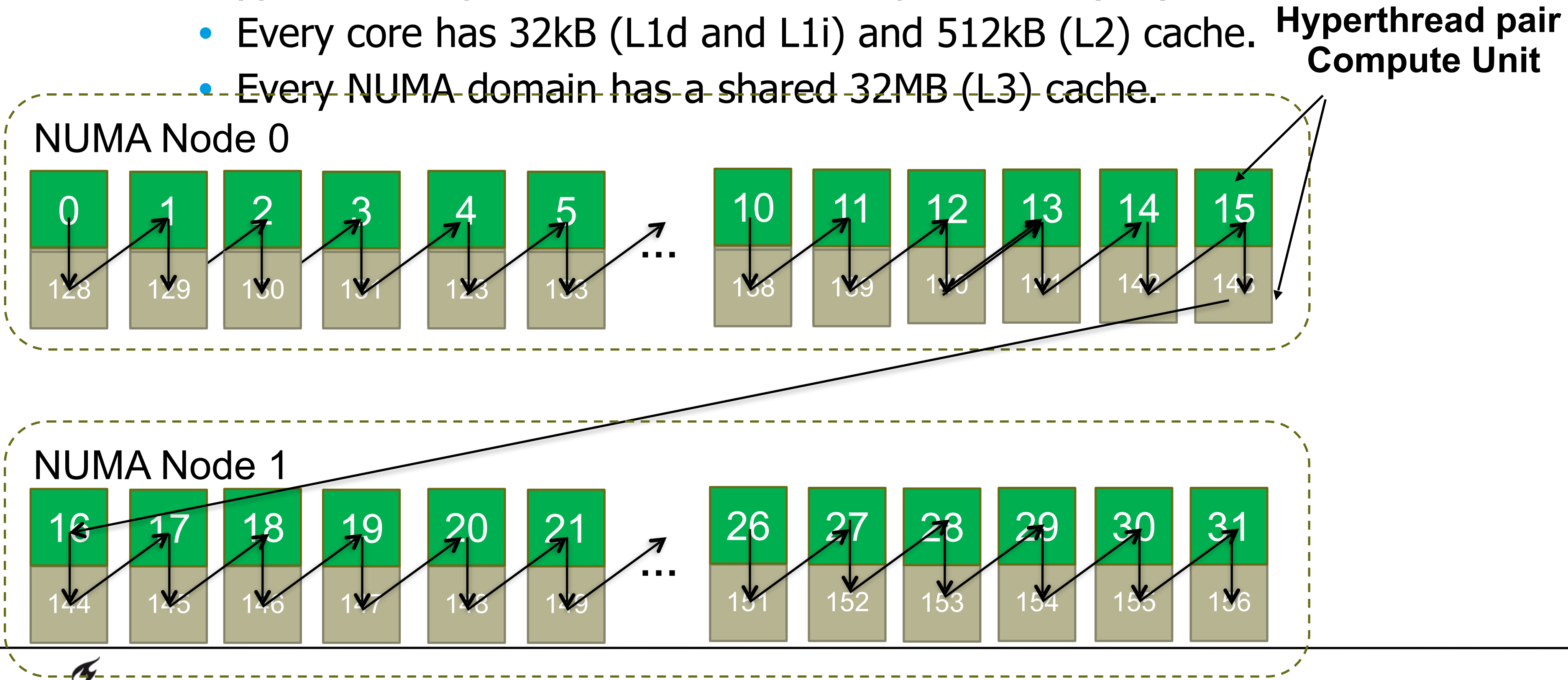
hyperthreads

- **Each NUMA domain contains 16 cores (2 hyperthreads)**
  - The numbering of the ‘actual cores’ is from 0-127 while the hyperthreads are numbered from 128-255.
  - A hyperthread pair is also called compute unit (CU) or core
    - Every core has 32kB (L1d and L1i) and 512kB (L2) cache.
    - Every NUMA domain has a shared 32MB (L3) cache.



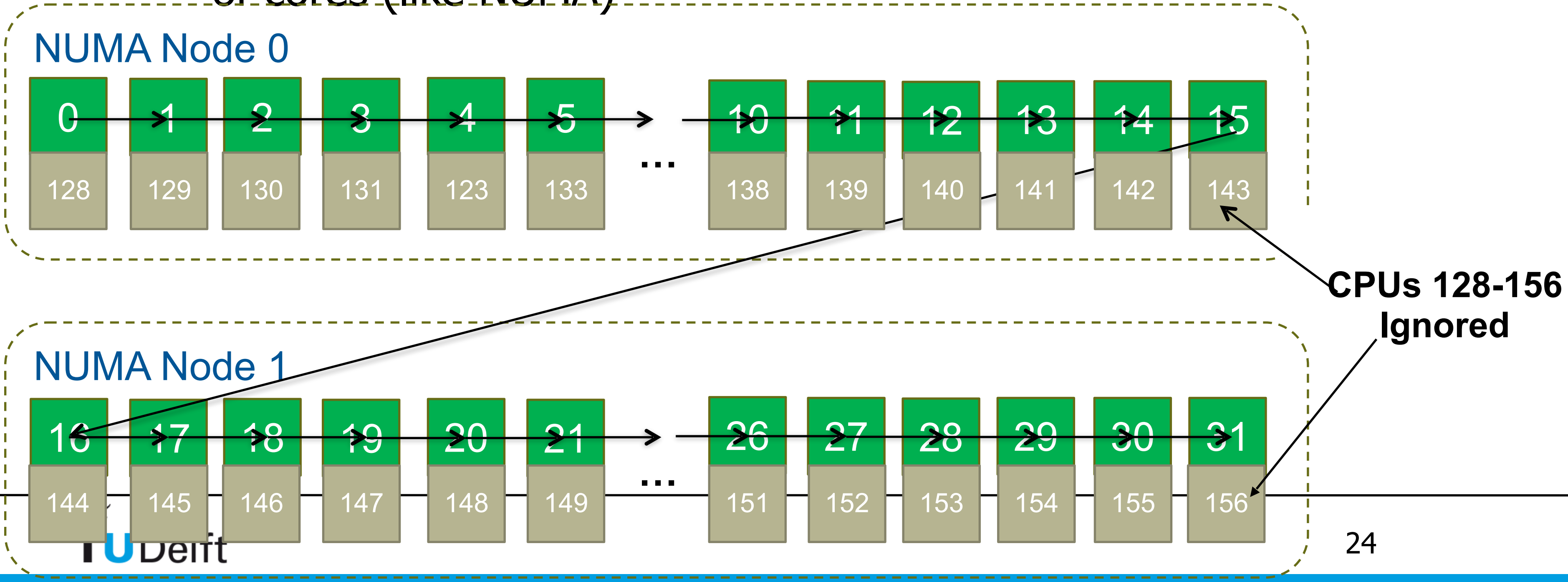
Hyperthreads and numbering (1)

- **Each NUMA domain contains 16 cores (2 hyperthreads)**
  - The numbering of the 'actual cores' is from 0-127 while the hyperthreads are numbered from 128-256.
  - A hyperthread pair is also called compute unit (CU) or core
    - Every core has 32kB (L1d and L1i) and 512kB (L2) cache.
    - Every NUMA domain has a shared 32MB (L3) cache.



Hyperthreads and numbering (2)

- **It is not mandatory to use the hyperthreads**
  - This can be achieved by `--hint=nomultithread` or an explicit binding lists.
  - The hyperthreads are still there but not utilized.
  - With or without hyperthreads, the software tasks and threads can be pinned to single cores or allowed to migrate on group of cores (like NUMA)

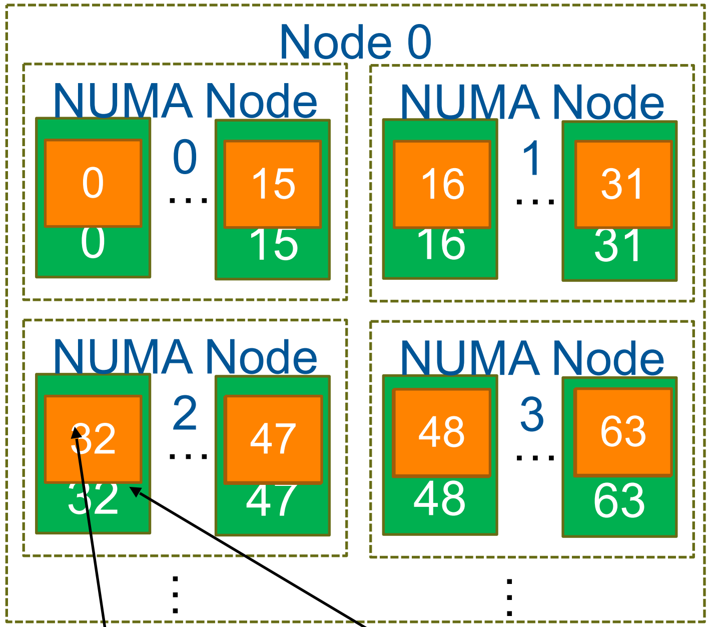


SLURM default binding

- **Assume that we have no threads and that we just run without specifying the binding?**
  - `> srun -n ${NPROCS}`

- **What happens?**
  - SLURM will spread the resources for you.
  - For SLURM this default is to fill up one node after another, whilst filling the NUMA regions of a node in alteration.
  - This is the same as using the srun command (we will discuss this more later)
    - `srun -n 128 --cpu-bind=cores --distribution=block:cyclic`
  - This may not be what is most desirable for your application.

Binding to ranks



1 Software task bound to 1 Hardware CPU

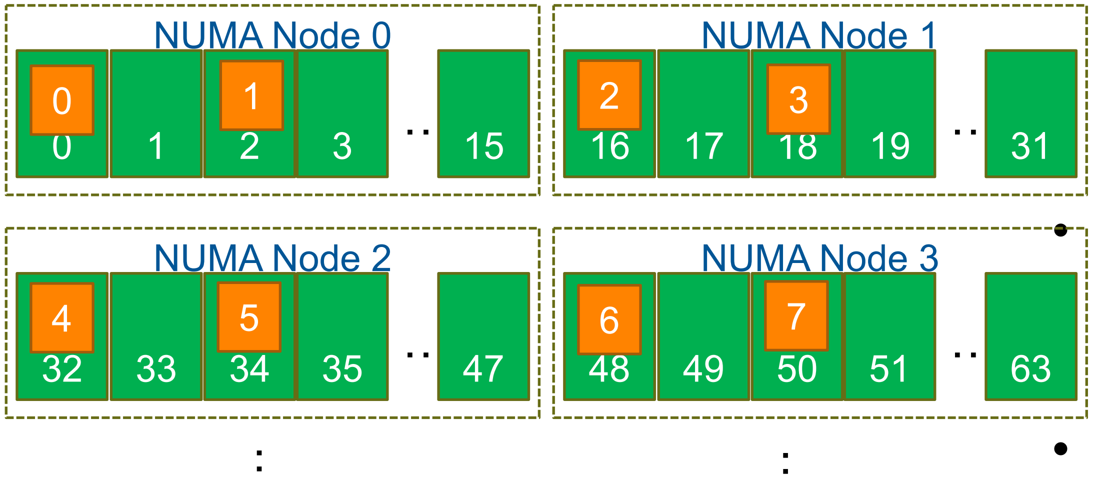
- **The user can bind tasks in “block mode”**
  - This will bind the task `i` to core `mod(i,128)`
- **The mapping is the same for all nodes.**

```
>>srun -n 128 --cpu-bind=rank
./${EXE}
```

Binding to ranks: xthi output

```
Hello from rank 0, thread 0, on nid001404. (core affinity = 0)
Hello from rank 1, thread 0, on nid001404. (core affinity = 1)
Hello from rank 2, thread 0, on nid001404. (core affinity = 2)
Hello from rank 3, thread 0, on nid001404. (core affinity = 3)
Hello from rank 4, thread 0, on nid001404. (core affinity = 4)
Hello from rank 5, thread 0, on nid001404. (core affinity = 5)
Hello from rank 6, thread 0, on nid001404. (core affinity = 6)
Hello from rank 7, thread 0, on nid001404. (core affinity = 7)
Hello from rank 8, thread 0, on nid001404. (core affinity = 8)
Hello from rank 9, thread 0, on nid001404. (core affinity = 9)
Hello from rank 10, thread 0, on nid001404. (core affinity = 10)
Hello from rank 11, thread 0, on nid001404. (core affinity = 11)
Hello from rank 12, thread 0, on nid001404. (core affinity = 12)
Hello from rank 13, thread 0, on nid001404. (core affinity = 13)
Hello from rank 14, thread 0, on nid001404. (core affinity = 14)
Hello from rank 15, thread 0, on nid001404. (core affinity = 15)
Hello from rank 16, thread 0, on nid001404. (core affinity = 16)
...
Hello from rank 126, thread 0, on nid001404. (core affinity = 126)
Hello from rank 127, thread 0, on nid001404. (core affinity = 127)
```

Custom binding with a map



- **The user can bind tasks explicitly to specific CPUs**
  - Binds each task to the CPUs listed map (in a round robin way if locations in map less than tasks)

The mapping is the same for all nodes.

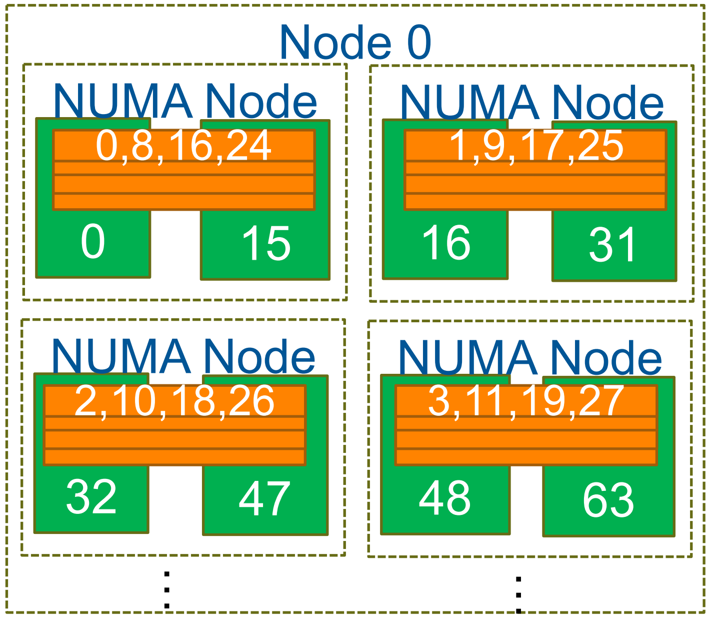
- **This can be useful when you have a very specific load distribution in mind for your application.**
- **Also, useful if you want to underpopulate a node to access more memory bandwidth per task.**

```
>>export bind=0,2,16,18,32,34,48,50
>>srun -n 8 --cpu-bind=map_cpu:${bind} ./${EXE}
```

TUDelft

Specifying a number of tasks per socket

- **The number of tasks per socket (or NUMA domain) can be limited.**
  - `--ntasks-per-socket=<>`

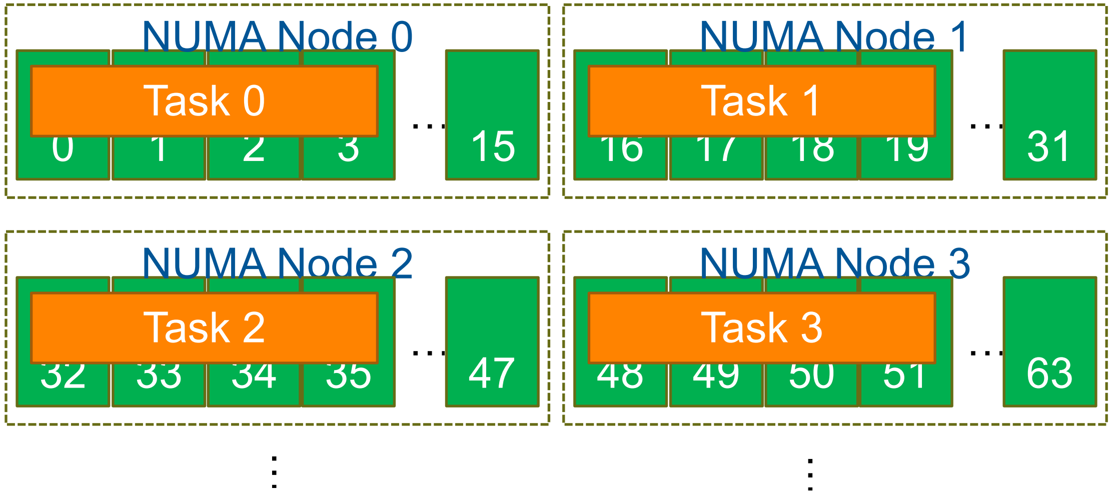


- **Places tasks on different NUMA domains in a round robin way.**
- **The tasks are allowed to migrate on the NUMA domain (actual cores and hyperthreads.)**
  - `--ntasks-per-socket=<>` seems to have a prevalence over `--ntasks-per-core=<>.`
  - If you do not use `--ntasks-per-socket=<>` there can be a distribution over NUMA nodes, but tasks/threads do not migrate over the entire domain.

```
>>srun -n 32 --ntasks-per-socket=16 ./${EXE
```

TUDelft

Hybrid binding



```
>>srun -n 8 –c 4 ./${EXE}
```

- You can use **–c** or **--cpus-per-task** to define how many threads you want per task
- Make sure that #threads divides the #core on a socket.
- Otherwise, a single task may spawn over 2 sockets.
- This can be fixed by adding `--ntasks-per-socket=<>` to force the task to another socket.

Openmp binding

Openmp binding

- **From OpenMP v4.0, OpenMP provides `environment` variables to specify how OpenMP threads should be bound to the system hardware.**

- **The variables are**
  - `OMP_PLACES`
  - `OMP_PROC_BIND`

- **Another useful variable to check for correctness is**
  - `OMP_DISPLAY_AFFINITY=TRUE`

TUDelft

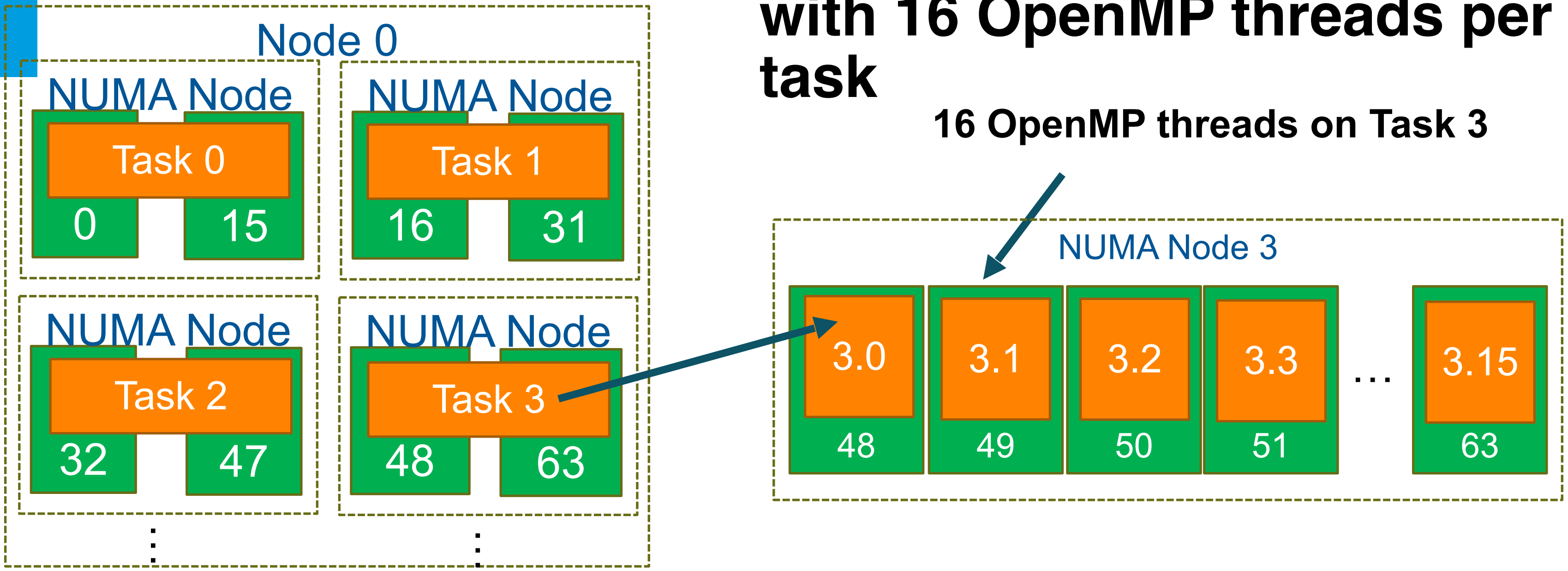
Omp_places

- A list of places that threads can be pinned on. The possible values are:
  - **threads**: Each place corresponds to a single hardware thread on the target machine.
  - **cores**: Each place corresponds to a single core (having one or more hardware threads) on the target machine.
  - **sockets**: Each place corresponds to a single socket (consisting of one or more cores) on the target machine.
  - A **list** with explicit values e.g., "{0:4}:4:4 = {0,1,2,3},{4,5,6,7},{8,9,10,11},{12,13,14,15}"
- Each "Place" defines a location where a thread can "float"

Omp_proc_bind

- Sets the binding of threads to processors.
  - **spread**: Bind threads as evenly distributed (spread) as possible.
  - **close**: Bind threads close to the master thread while still distributing threads for load balancing.
  - **master**: Bind threads to the same place as the master thread.
  - **false**: turns off OMP binding

Combining MPI tasks and OpenMP threads: OMP binding

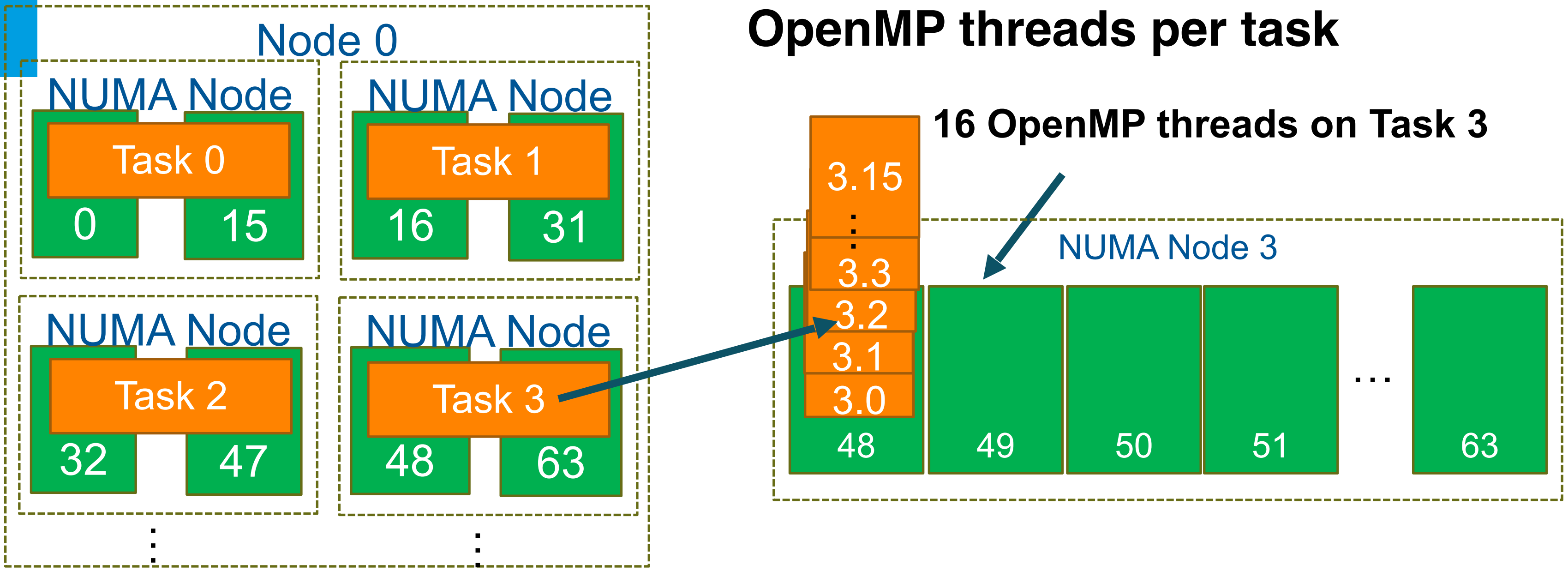


- **Here we specify 8 MPI tasks with 16 OpenMP threads per task**

```
>>export
OMP_PROC_BIND={true,close,spread}
>>export OMP_NUM_THREADS=16
>>srun -n 8 -c ${OMP_NUM_THREADS} ./
${EXE}
```

TU Delft

Combining MPI tasks and OpenMP threads: OMP binding



- **Here we specify 8 MPI tasks with 16 OpenMP threads per task**

```
>>export OMP_PROC_BIND=master
>>export OMP_NUM_THREADS=16
>> srun -n 8 -c ${OMP_NUM_THREADS}
./${EXE}
```

TU Delft

Openmp vs slurm mechanisms

- **OpenMP is a standard**
  - Will work with each compiler Run Time Environment (RTE)
- **However, OpenMP knows nothing about the MPI ranks**
  - Still need SLURM (or another batch system) or an MPI implementation to distribute the ranks
- **The OpenMP standard added an abstraction layer which can be useful when defining a complex layout**

SLURM Task distribution (level 1)

- **To control the distribution of the MPI ranks (tasks) across nodes, use the `'--distribution/-m'` argument to `srun`.**
  - `--distribution=plane=X –nodes=N`
    Will distribute N/X blocks cyclic, each of the size of X tasks
    Note : No node will be empty of tasks
    `srun --nodes=5 -n 12  -m plane=5 ./{EXE}`
    creates the distribution : 0 0 0 0 0 1 1 1 1 2 3 4

  - `--distribution=block`
    Will distribute tasks such that consecutive tasks share a node :
    `srun --nodes=5 -n 12  --distribution=block ./{EXE}`
    creates the distribution : 0 0 0 1 1 1 2 2 3 3 4 4

  - `--distribution=cyclic`
    Will distribute tasks such that consecutive tasks are distributed over consecutive nodes (round robin) :
    `srun --nodes=5 -n 12  --distribution=cyclic ./{EXE}`
    creates the distribution : 0 1 2 3 4 0 1 2 3 4 0 1

TU Delft

SLURM Task distribution (level 2)

- **For the second distribution method, the ranks collected in a node in the first distribution step, can be distributed over the sockets/NUMA nodes.**

  - `--distribution=[block|cyclic]:block`
    This will distribute allocated CPUs for binding to tasks such that consecutive tasks share a socket, before moving to the next consecutive socket.

  - `--distribution=[block|cyclic]:cyclic`
    This will distribute allocated CPUs for binding to a given task such that consecutive tasks are distributed over consecutive NUMA regions (round robin). Any task requiring more than one CPU will be given those from a single NUMA region

  - `--distribution=[block|cyclic]:fcyclic`
    Same as cyclic but tasks requiring more than one CPU will have these allocated cyclically across NUMA regions.

Setting and controlling affinity, slurm-srun

- **Other affinity-related, potentially useful `srun` options:**
  - `--exclusive`; A given job has exclusive access to a node's resources, no other jobs have access
  - `--mem_bind=[{quiet,verbose}],type`; Bind tasks to memory For example --mem_bind=local, will bind each task to use only its own NUMA node memory
  - Application hints:
    - `--hint=memory_bound`, maximize memory bandwidth and use one core per socket
    - `--hint=compute_bound` , maximize compute and use all cores per socket

Reporting binding

- **srun verbose**
  ```
  srun --cpu-bind=verbose,….
  ```
- **MPI**
  ```
  MPICH_CPUMASK_DISPLAY=1
  ```
- **OpenMP**
  ```
  OMP_DISPLAY_AFFINITY=TRUE
  ```

Exercise: Affinity

- Just to play around and learn.