Hewlett Packard Enterprise

Advanced Placement

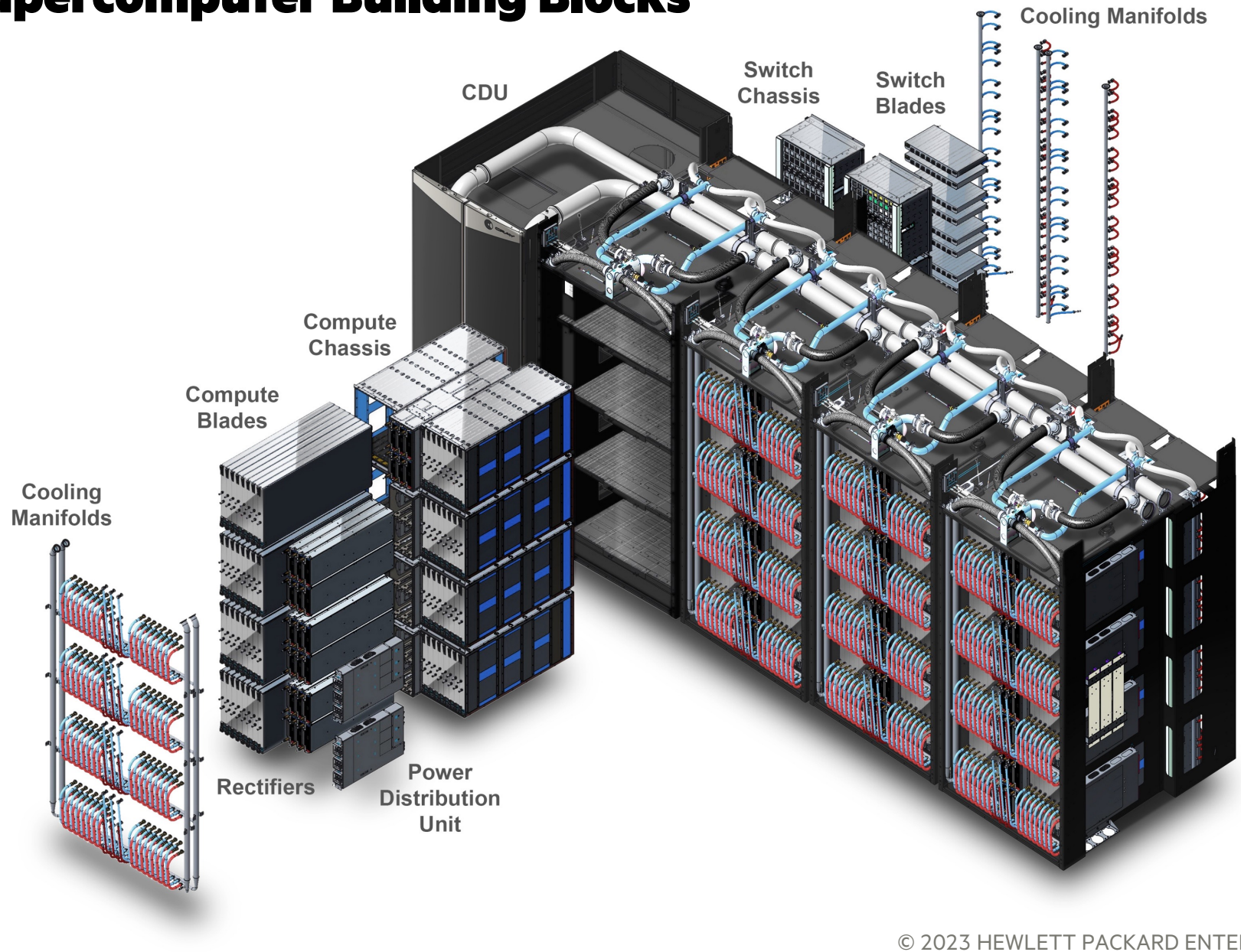
Introduction to GPU - PDC

Oct 12–13, 2023

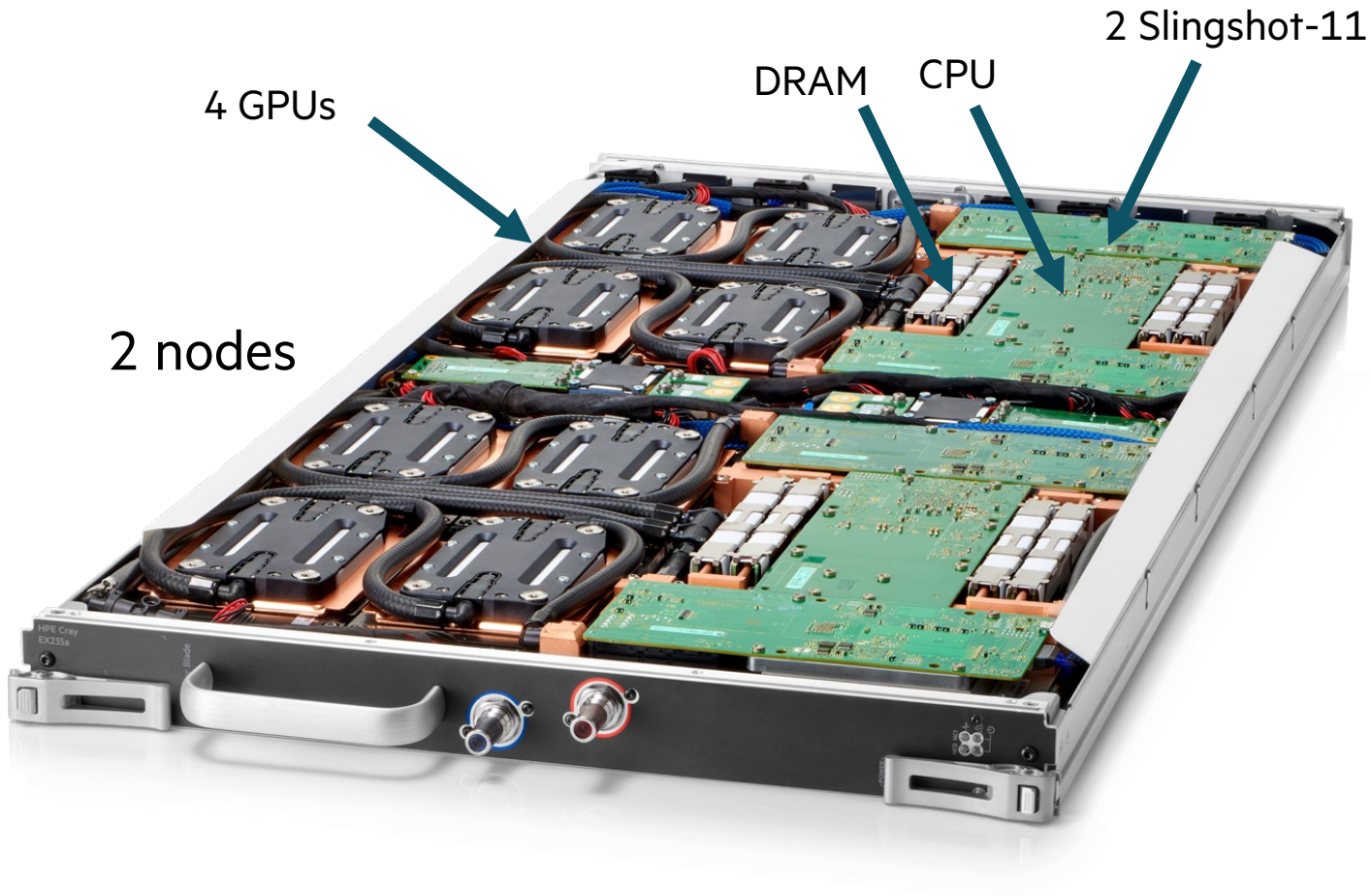
Agenda

- Hardware context and terminology
- Mapping applications to hardware
  - Task binding with Slurm
  - Task distribution with Slurm
- Examples of using the Slurm Batch system
  - OpenMP support for thread binding
  - Controls for job placement (CPU/GPU/NIC)
  - Bit masks
- Final remarks

Cray EX Supercomputer Building Blocks



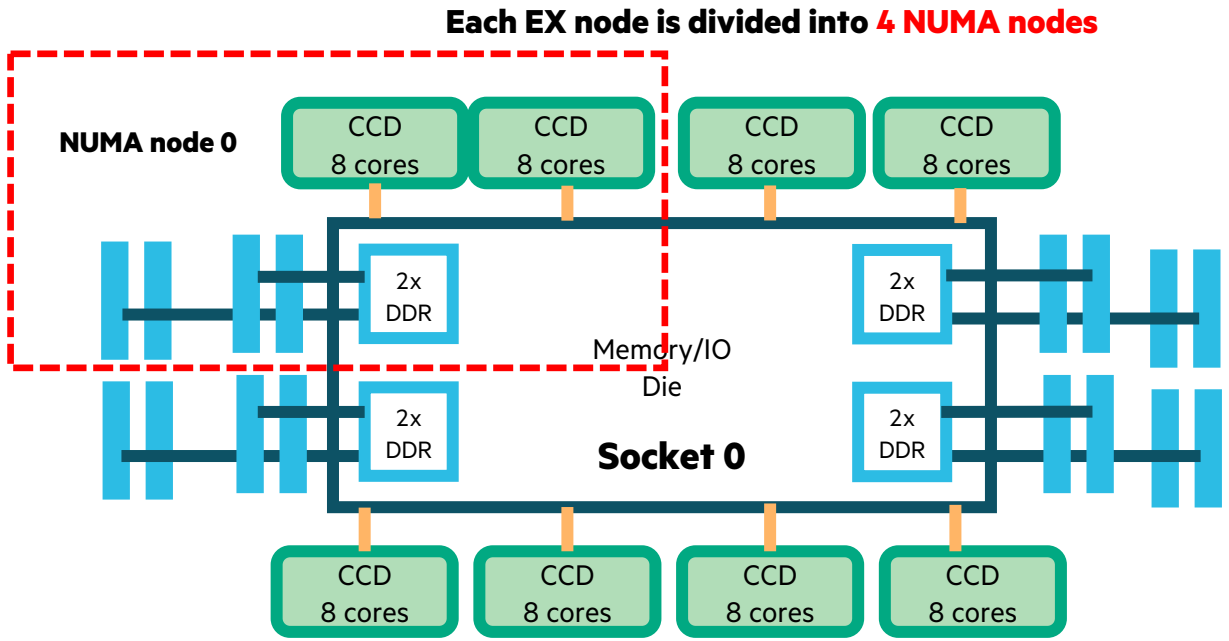
COMPUTE BLADE ARCHITECTURE (GPU PARTITION)



Each node:

- AMD EPYC 7A53 “Optimized 3rd Gen EPYC” 64-Core Processor, 2.00 GHz
- 512 GB DDR4 memory
- 4x AMD MI-250X GPU
- Each GPU connected to a Slingshot 200Gb/s NIC

AMD CPU



AMD EPYC 7A53

Base **clock 2.00 GHz**

64 cores, 128 hardware threads

- 8 CCDs of 8 cores

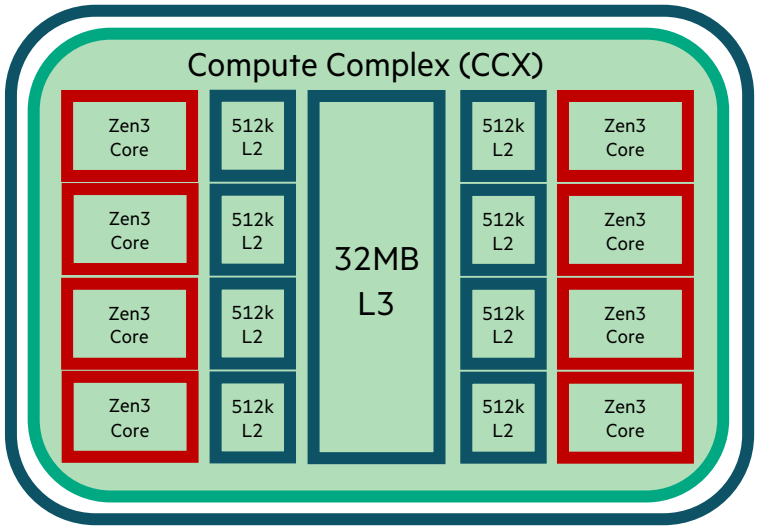
256MB L3 cache in total

8 channel DDR4 3200MHz, 204.8 GB/s peak b/w

Configured as **4 NUMA nodes**

Vector support: AVX2

Compute Complex Die (CCD)



Compute Complex Dies (CCDs)

- host cores and L2/L3 cache
  - L1 cache 32kB / core
  - L2 cache 512kB / core
  - L3 cache 32MB / 8-cores

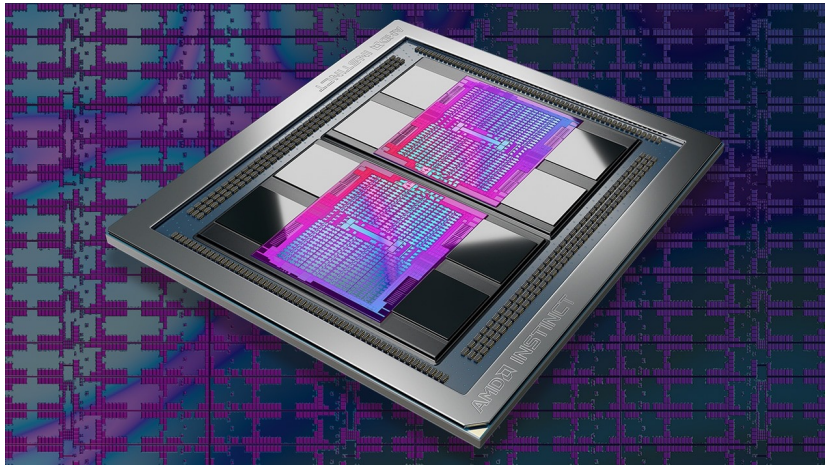
Mi250X GPU Architecture

- Two compute dies (Graphics Compute Dies (GCDs))
  - Interconnected with 200 GB/s per direction

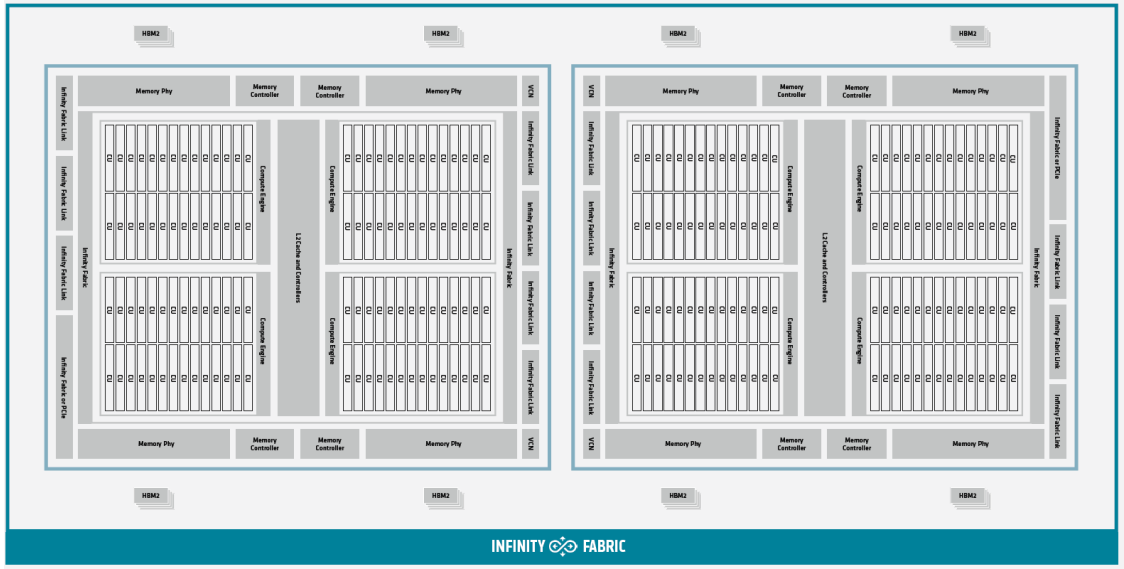
- Dedicated Memory (HBM2e) Size: 128 GB
  - High bandwidth device memory (up to 3.2 TB/s)
  - Memory Clock: 1.6 GHz

- AMD CDNA2 Architecture
  - 110 Compute Units (CU) per each die = 220 CUs
  - 64 SIMD threads per each CU = 14080 Stream Processors
  - Peak FP64/FP32 Vector: 47.9 TFLOPS
  - Peak FP64/FP32 Matrix: 95.7 TFLOPS
  - Total L2 cache: 8 MB per each die (64kB per CU)
  - Frequency: up to 1700 MHz
  - Max power: up to 560 Watts

- Memory coherency with CPU



Source: https://www.amd.com/en/products/server-accelerators/instinct-mi250

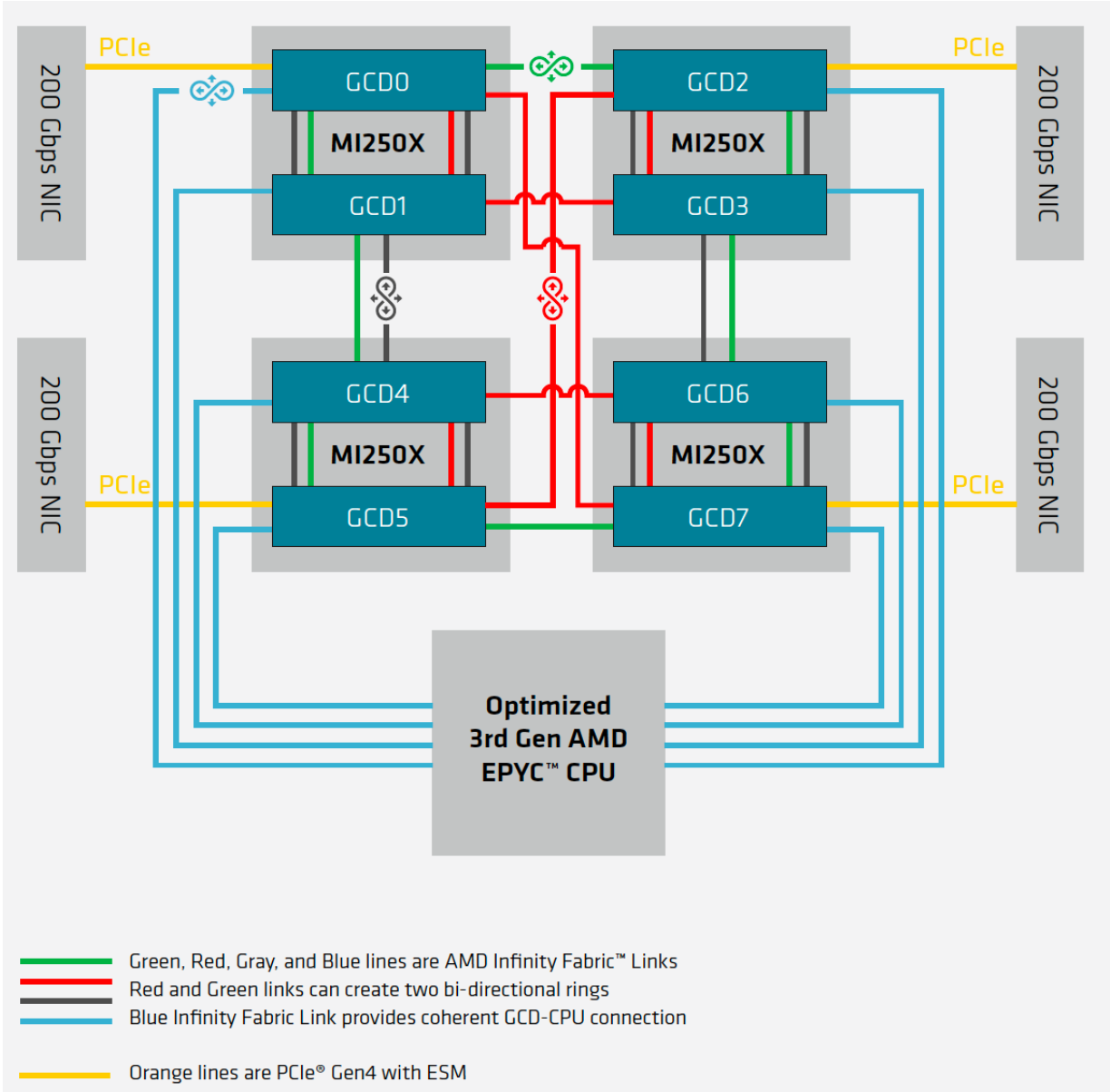


Source: (https://www.amd.com/system/files/documents/amd-cdna2-white-paper.pdf)

NODE ARCHITECTURE (LUMI-G)

- The programmer can think of the 8 GCDs as 8 separate GPUs, each having 64 GB of high-bandwidth memory (HBM2E)
- The CPU is connected to each GCD via Infinity Fabric CPU-GPU, allowing a peak host-to-device (H2D) and device-to-host (D2H) bandwidth of 36+36 GB/s
  - Coherent memory CPU-GPU
- The 2 GCDs on the same MI250X are connected with Infinity Fabric GPU-GPU
- The GCDs on different MI250X are connected with Infinity Fabric GPU-GPU in the arrangement shown in the diagram on the right, where the peak bandwidth ranges from 50-100 GB/s based on the number of Infinity Fabric connections between individual GCDs

Source: https://www.amd.com/system/files/documents/amd-cdna2-white-paper.pdf



Hardware : command and tools

A Quick Recap – Glossary of Terms

- **Hardware** - This terminology is used to cover hardware from multiple vendors
  - **Socket**
    The hardware you can touch and insert into the mother board
  - **Core**
    The individual unit of hardware for processing, part of the CPU. This can be called a **compute unit** (CU)
  - **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**.

- **Software** - Different software approaches also use different naming convention.
  - **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)

Getting Node Information

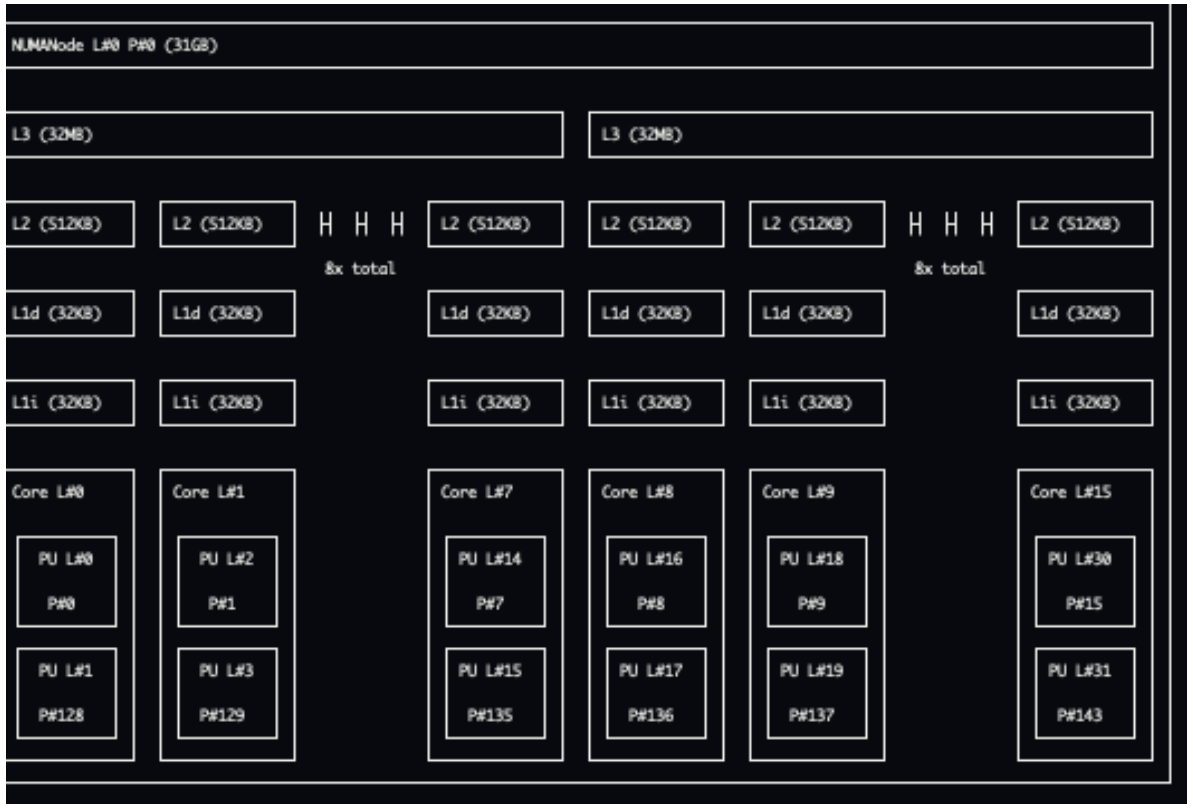
```
lscpu | grep -Ei "CPU\(s\)|Thread|Core\(s\)
 |Socket\(s\)|Numa|Model\ name| MHz|cache"
```

CPU(s):	128
On-line CPU(s) list:	0-127
Thread(s) per core:	2
Core(s) per socket:	64
Socket(s):	1
NUMA node(s):	4
Model name:	AMD EPYC 7A53
CPU MHz:	1925.019
CPU max MHz:	3541.0149
CPU min MHz:	1500.0000
L1d cache:	32K
L1i cache:	32K
L2 cache:	512K
L3 cache:	32768K
NUMA node0 CPU(s):	0-15,64-79
NUMA node1 CPU(s):	16-31,80-95
NUMA node2 CPU(s):	32-47,96-111
NUMA node3 CPU(s):	48-63,112-127

- Check **`/proc/cpuinfo`** ON the compute nodes
- If **`lscpu`** is installed on a system, it will list the configuration
- Hyperthreading (aka SMT) is turned ON
  - From a binding point of view, here we have **CORES=CPU**
- Also try the command **`numactl --hardware`**

LSTOPO : GRAPHICAL AND ASCII VIEW

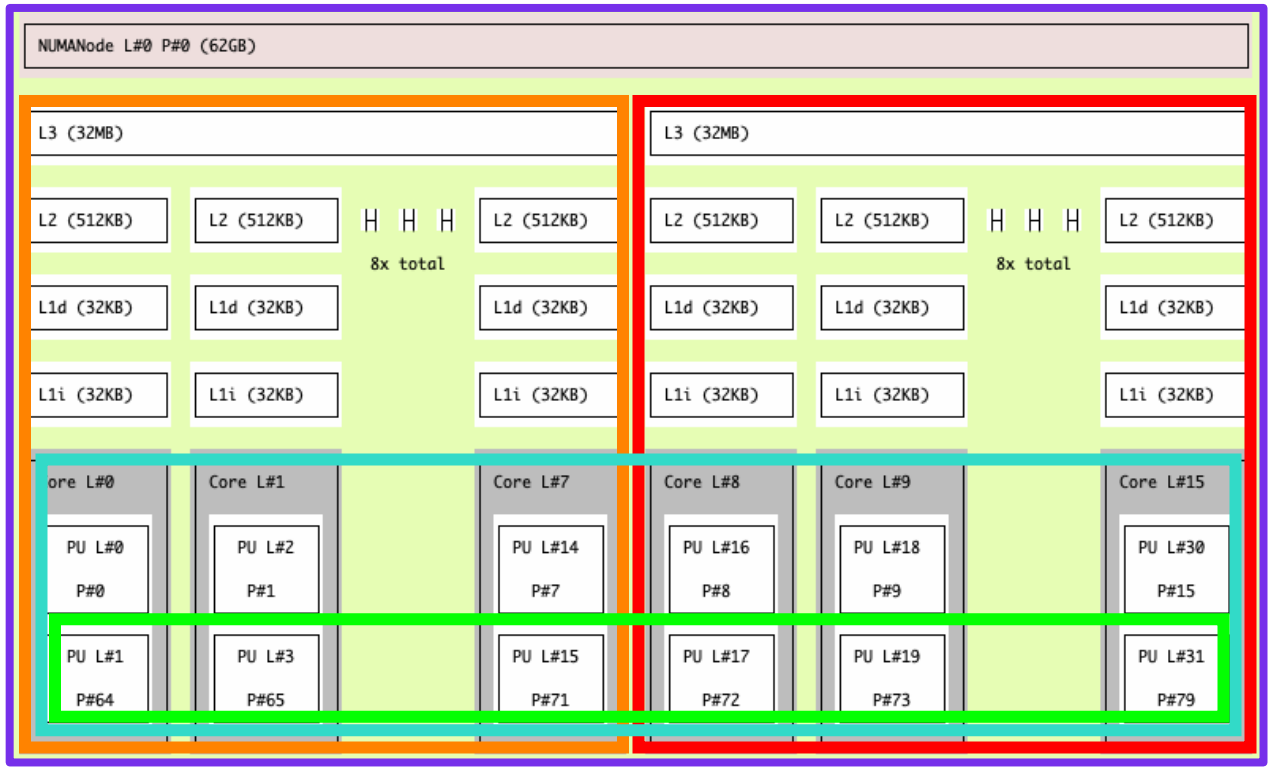
`lstopo --output-format svg -v --no-io > cpu.svg`



`lstopo-no-graphics -.ascii --only pu`

GETTING NODE INFORMATION

```
numactl --hardware | grep -E "nodes|cpus"
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 64 65 66 67 68 69 70 71 72 73 74 75 76 ...
node 1 cpus: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 80 81 82 83 84 85 86 87 88 89 ...
...
```



1 socket = 4 numa nodes

Each **NUMA NODE** (4 in total) has

- 16 physical core
- Each core has an hyperthread
- Cores are divided in **2 CCD** (**orange**/**red**)

```
lscpu | grep NUMA
NUMA node(s):          4
NUMA node0 CPU(s):     0-15,64-79
NUMA node1 CPU(s):     16-31,80-95
NUMA node2 CPU(s):     32-47,96-111
NUMA node3 CPU(s):     48-63,112-127
```

GPUS TO NUMA DOMAINS MAPPING

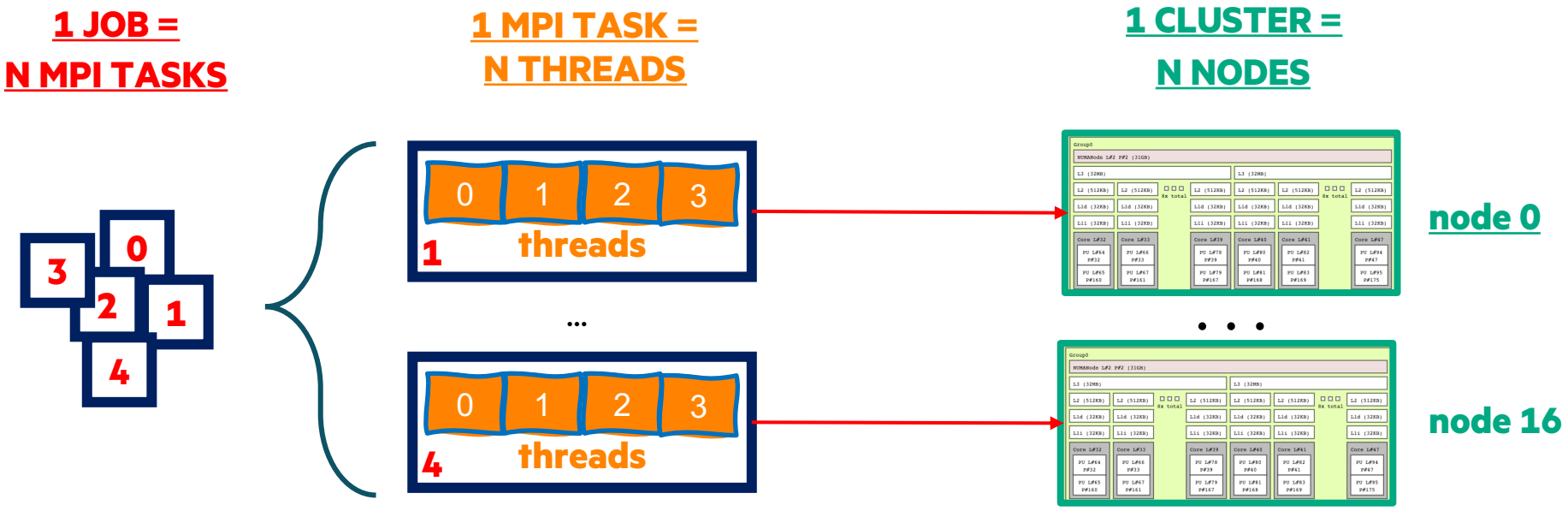
- GPUs are associated to NUMA nodes
- Can use rocm-smi to see the topology

```
> srun --nodes=1 -p <partition> -A <your_project> -t "00:02:00" --ntasks=1 --gres=gpu:8 rocm-smi --showtopo
…
================================== Numa Nodes ==================================
GPU[0] : (Topology) Numa Node: 3
GPU[0] : (Topology) Numa Affinity: 3
GPU[1] : (Topology) Numa Node: 3
GPU[1] : (Topology) Numa Affinity: 3
GPU[2] : (Topology) Numa Node: 1
GPU[2] : (Topology) Numa Affinity: 1
GPU[3] : (Topology) Numa Node: 1
GPU[3] : (Topology) Numa Affinity: 1
GPU[4] : (Topology) Numa Node: 0
GPU[4] : (Topology) Numa Affinity: 0
GPU[5] : (Topology) Numa Node: 0
GPU[5] : (Topology) Numa Affinity: 0
GPU[6] : (Topology) Numa Node: 2
GPU[6] : (Topology) Numa Affinity: 2
GPU[7] : (Topology) Numa Node: 2
GPU[7] : (Topology) Numa Affinity: 2
============================== End of ROCm SMI Log ==============================
```

Job Task Placement: Mapping Application to Hardware

Goal of the presentation

- **Process distribution** : distribution of MPI processes among the nodes.
- **Process binding** : pin/attache the MPI processes (and their threads) to one or many cores.



What is “binding”?

- When a process executes it does so with affinity to a set of CPUs that it can execute on. We call this the **binding** of the process to the CPUs.

- **The concept of mapping tasks or threads to hardware is crucial for optimal performance :**
  - Memory locality (minimize the data movement internally on the processor)
  - Make sure threads runs only on one CPU
    - To reduce OS costs when moving it
    - To make sure no CPU is oversubscribed

- Improper process/thread affinity could **slow down code performance** significantly
  - It can be hard to spot performance problems relating to binding
  - In some cases, this can be wanted.

BINDING only makes sense if the node is reserved in **--exclusive** mode

CPU binding

- You should be aware **WHO is doing** the binding for you!
- **Various software** components may try do this for you
  - WLM (SLURM, ALPS, …)
  - MPI (MVAPICH, …)
  - Compiler (CCE,GNU,…)
  - OpenMP

- **Slurm** allows users to modify the process binding by using :

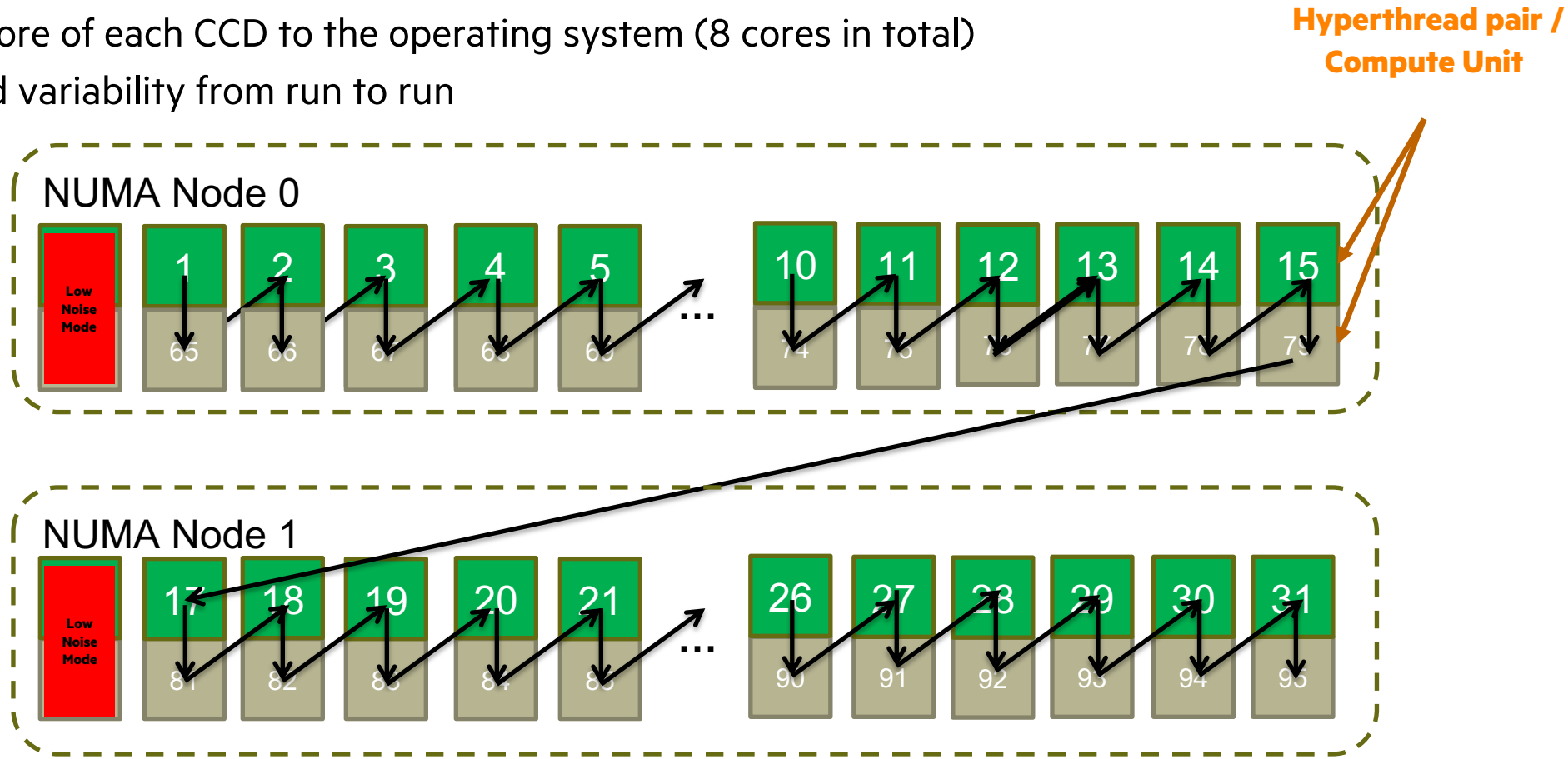
  `--cpu_bind``=<cores|sockets>`

  `--distribution``=<block|cyclic>`

- **Verify binding by using applications such as:**
  - **CPU- Xthi**
    https://support.hpe.com/hpesc/public/docDisplay?docId=a00114008en_us&docLocale=en_US&page=Run_an_OpenMP_Application.html
  - **GPU – Hello Jobstep**
    https://code.ornl.gov/olcf/hello_jobstep/-/tree/master

Hyperthreads and numbering (1)

- By default = multithreads **<u>are enabled</u>**
- The GPU nodes have the low-noise mode activated.
  - This mode reserve 1 core of each CCD to the operating system (8 cores in total)
  - Helps reduce jitter and variability from run to run



Task Binding with Slurm

Task Binding with Slurm

- Slurm allows users to modify the process binding by using the **`--cpu-bind=<mode>`** option to **`srun`** (or the env. variable **`SLURM_CPU_BIND`**), e.g., `none`, `rank`, `cores`, `sockets`
  - Can use **`--cpu-bind=verbose,<mode>`** reports binding before task runs

- There are several different ways of distributing MPI ranks over cores via the Slurm option **`--distribution=<mode>`**, e.g., **`block`**, **`cyclic`**, **`plane`**
  - Multi-level control: tasks to node, tasks to sockets, tasks to cores

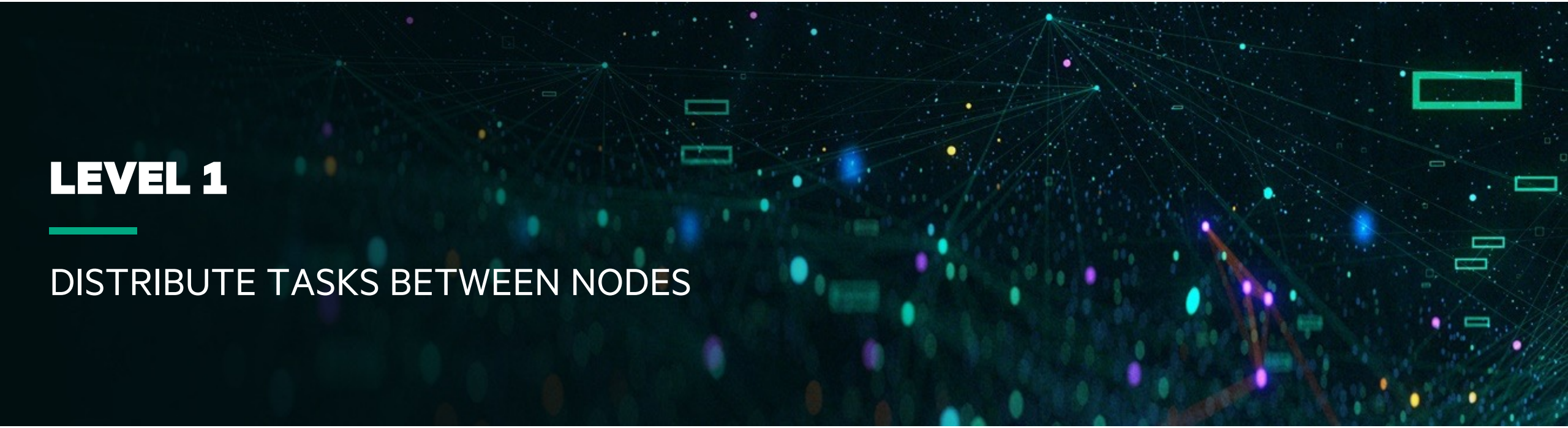
- By default, Slurm options used are

```
--cpu-bind=threads --distribution=block:cyclic
```

- For all examples we are setting **`--hint=nomultithread`**

SLURM Task distribution

- The **`-m`** or **`--distribution`** argument to `srun` controls the distribution of MPI tasks across:
  - Nodes (level 1) e.g. **`-m block`**
  - Sockets (level 2) e.g. **`-m block:cyclic`**
  - Cores (level 3) e.g. **`-m block:cyclic:fcylic`**

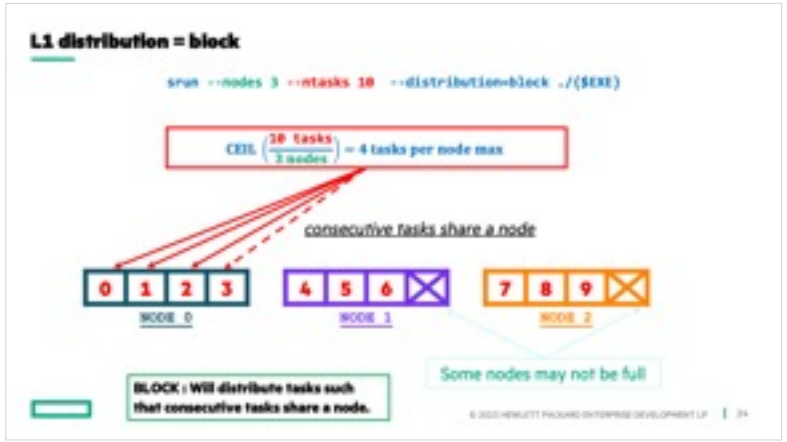


LEVEL 1

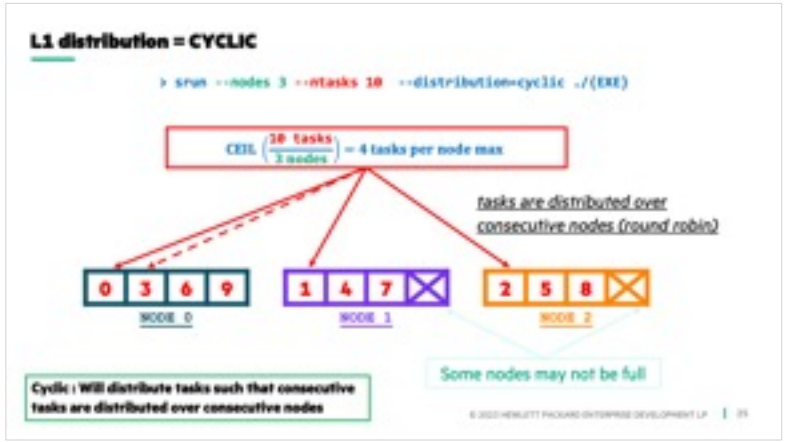
DISTRIBUTE TASKS BETWEEN NODES

LEVEL 1 = DISTRIBUTE TASKS BETWEEN NODES

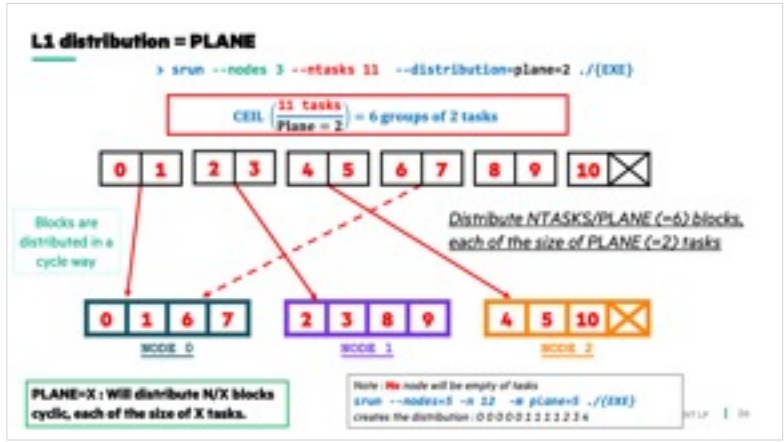
- To control the distribution of the MPI ranks (tasks) across nodes
- In *srun* : use the '`--distribution/-m`' with either { block | cyclic | plane=<size> }



`--distribution=block`



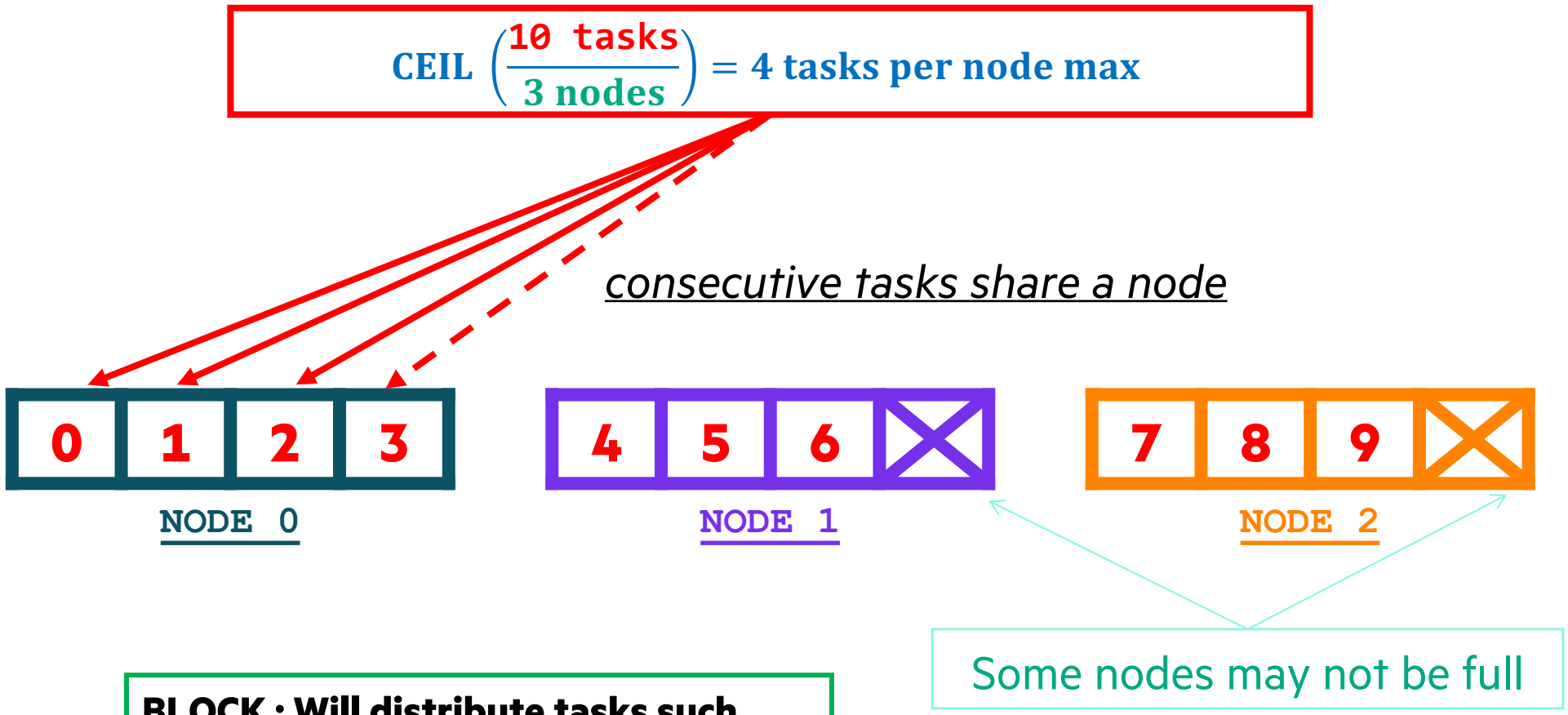
`--distribution=cyclic`



`--distribution=plane=X`

L1 distribution = block

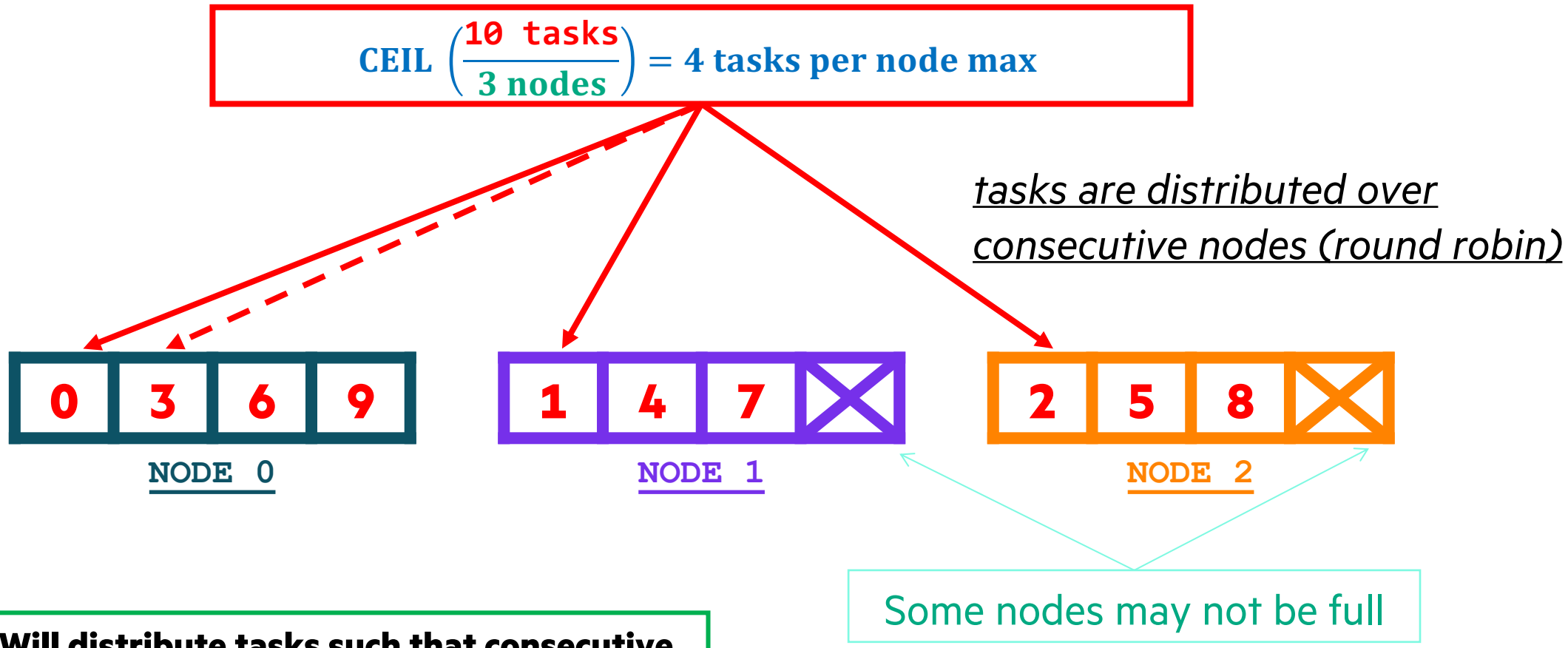
```
srun --nodes 3 --ntasks 10  --distribution=block ./{$EXE}
```



BLOCK : Will distribute tasks such that consecutive tasks share a node.

L1 distribution = CYCLIC

```
> srun --nodes 3 --ntasks 10  --distribution=cyclic ./{EXE}
```

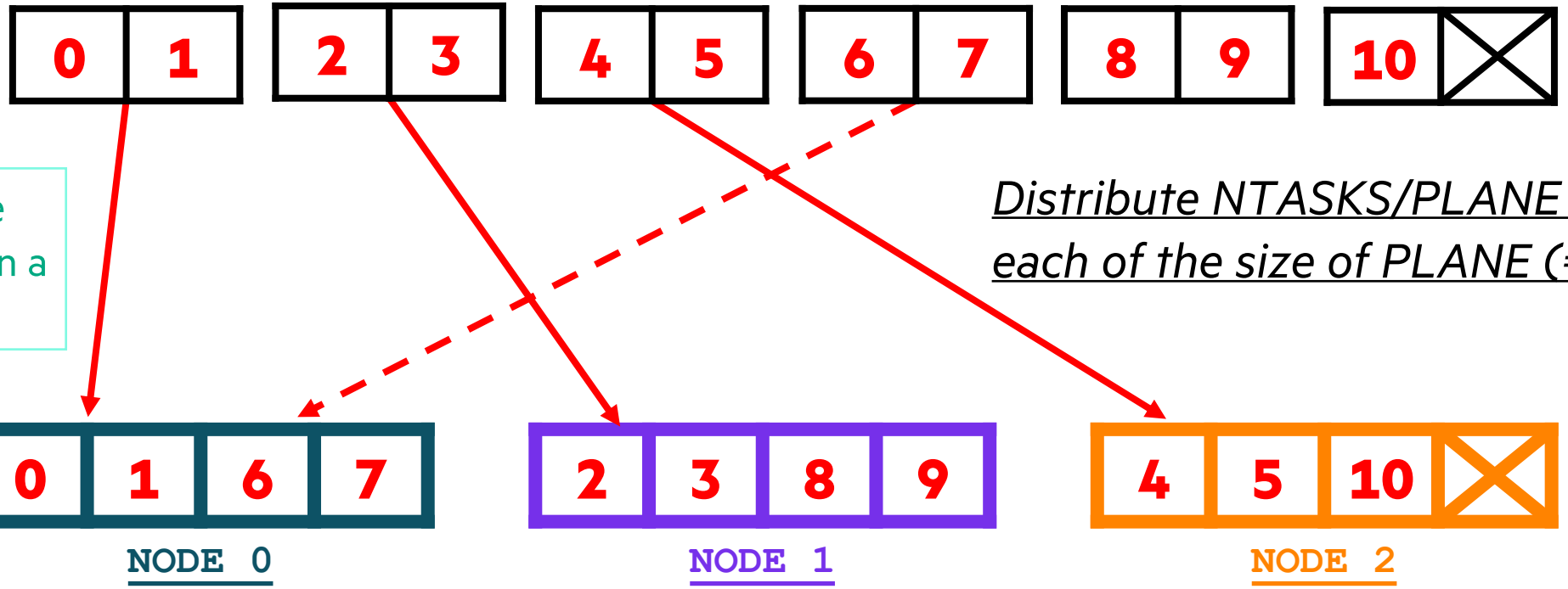


Cyclic : Will distribute tasks such that consecutive tasks are distributed over consecutive nodes

L1 distribution = PLANE

```
> srun --nodes 3 --ntasks 11  --distribution=plane=2 ./{EXE}
```

$$\text{CEIL}\left(\frac{11 \text{ tasks}}{\text{Plane} = 2}\right) = 6 \text{ groups of 2 tasks}$$



Blocks are distributed in a cycle way

Distribute NTASKS/PLANE (=6) blocks, each of the size of PLANE (=2) tasks

PLANE=X : Will distribute N/X blocks cyclic, each of the size of X tasks.

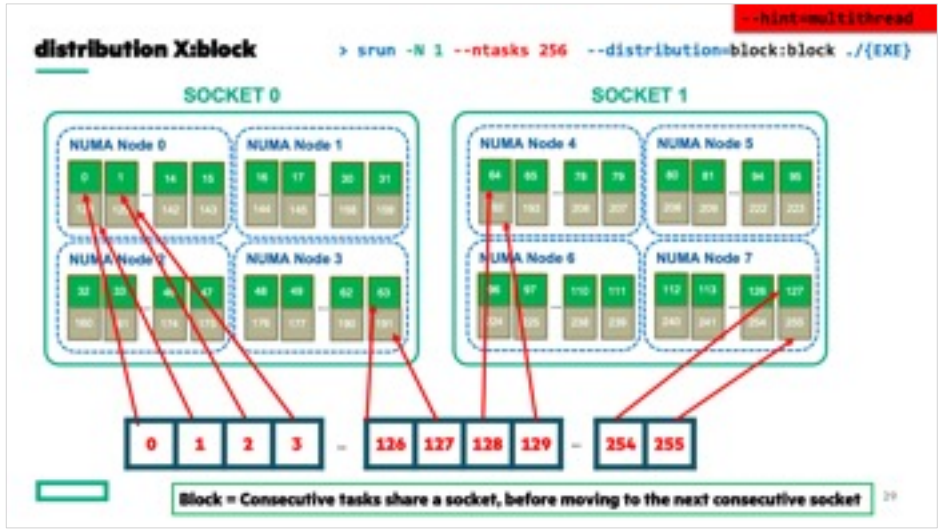
*Note : **No** node will be empty of tasks*
`srun --nodes=5 -n 12  -m plane=5 ./{EXE}`
creates the distribution : 0 0 0 0 0 1 1 1 1 2 3 4

LEVEL 2

Bind task to hardware

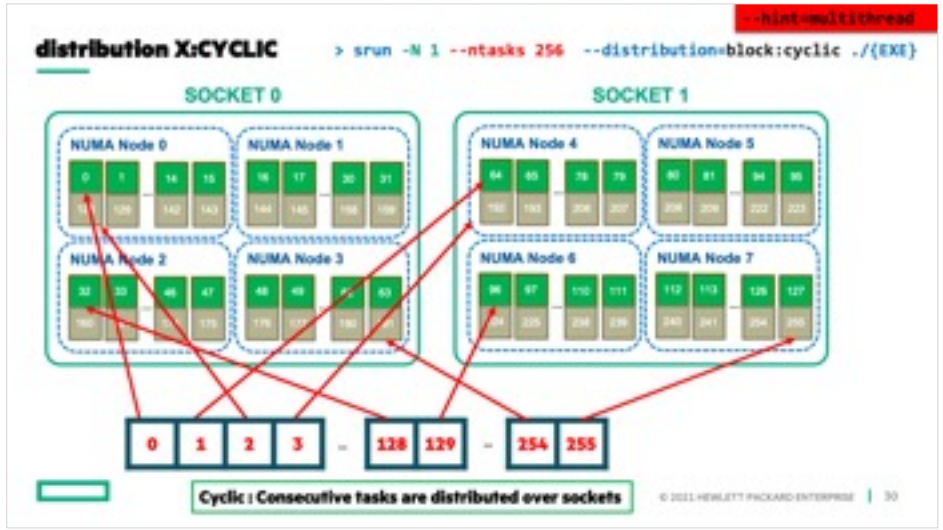
SLURM Task distribution (level 2)

- **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`

consecutive tasks share a socket



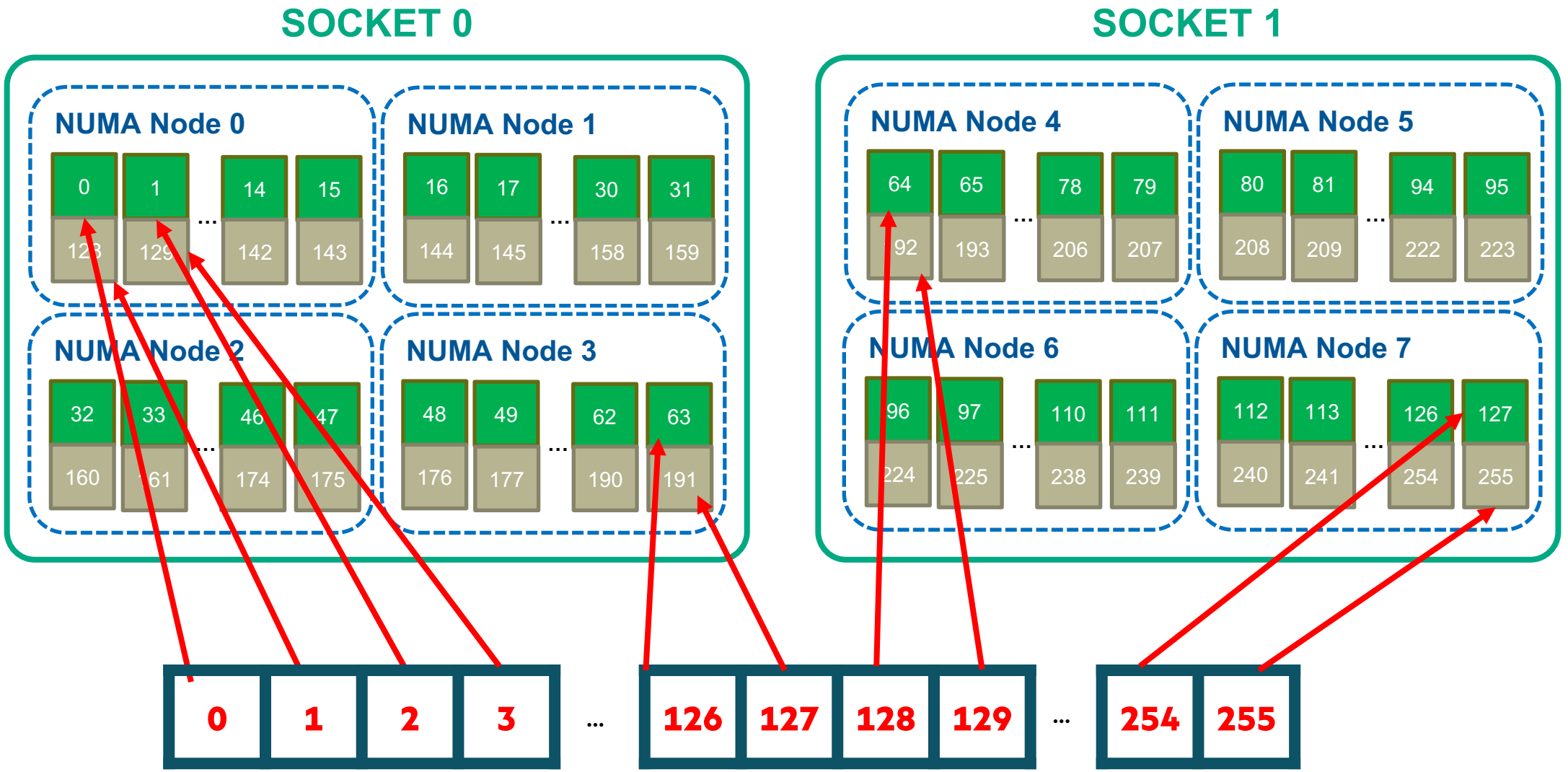
`--distribution=[block|cyclic]:cyclic`

consecutive tasks are distributed over NUMA

--hint=multithread

distribution X:block

```
> srun -N 1 --ntasks 256  --distribution=block:block ./{EXE}
```

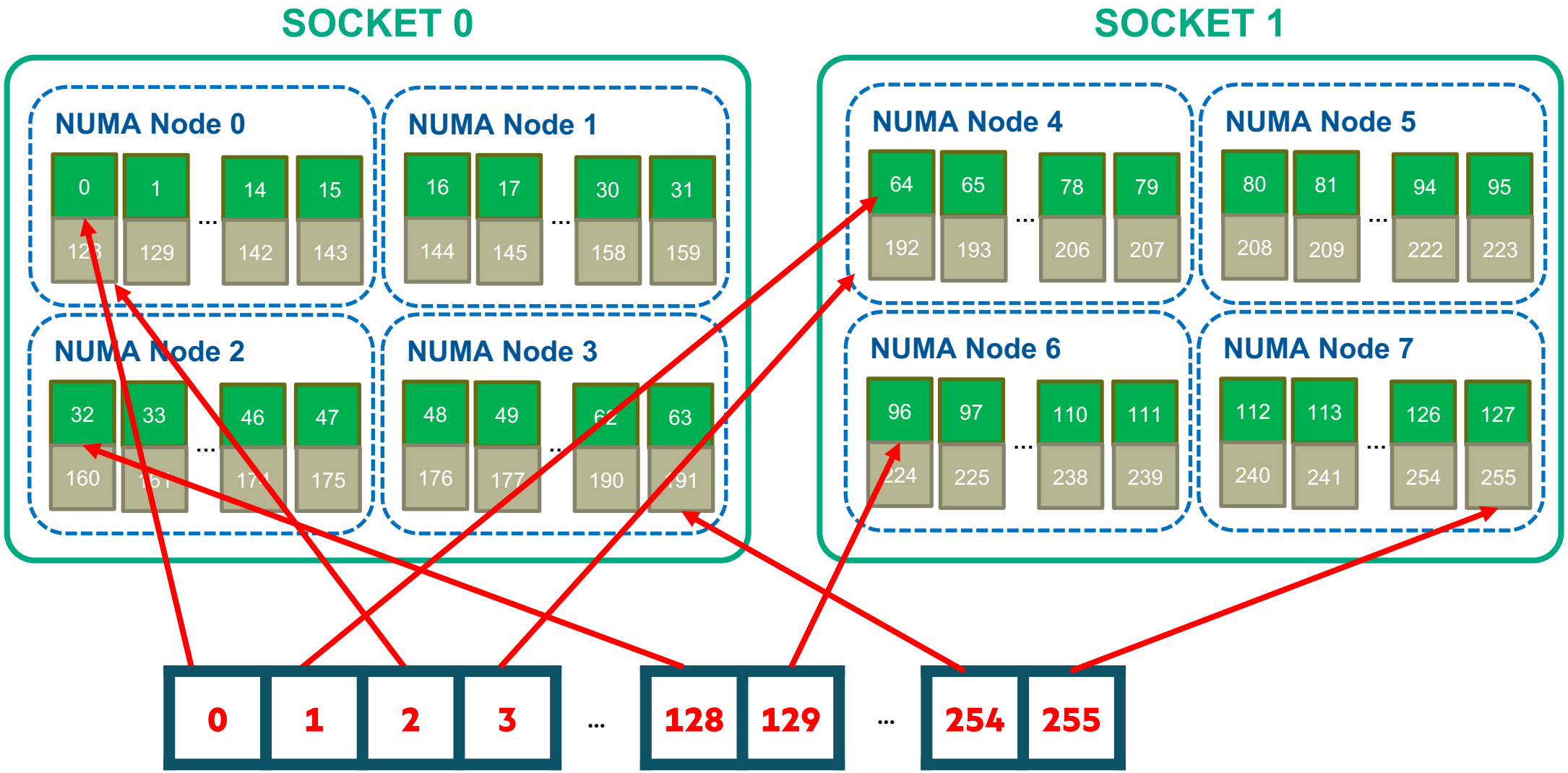


Block = Consecutive tasks share a socket, before moving to the next consecutive socket

--hint=multithread

distribution X:CYCLIC

```
> srun -N 1 --ntasks 256  --distribution=block:cyclic ./{EXE}
```

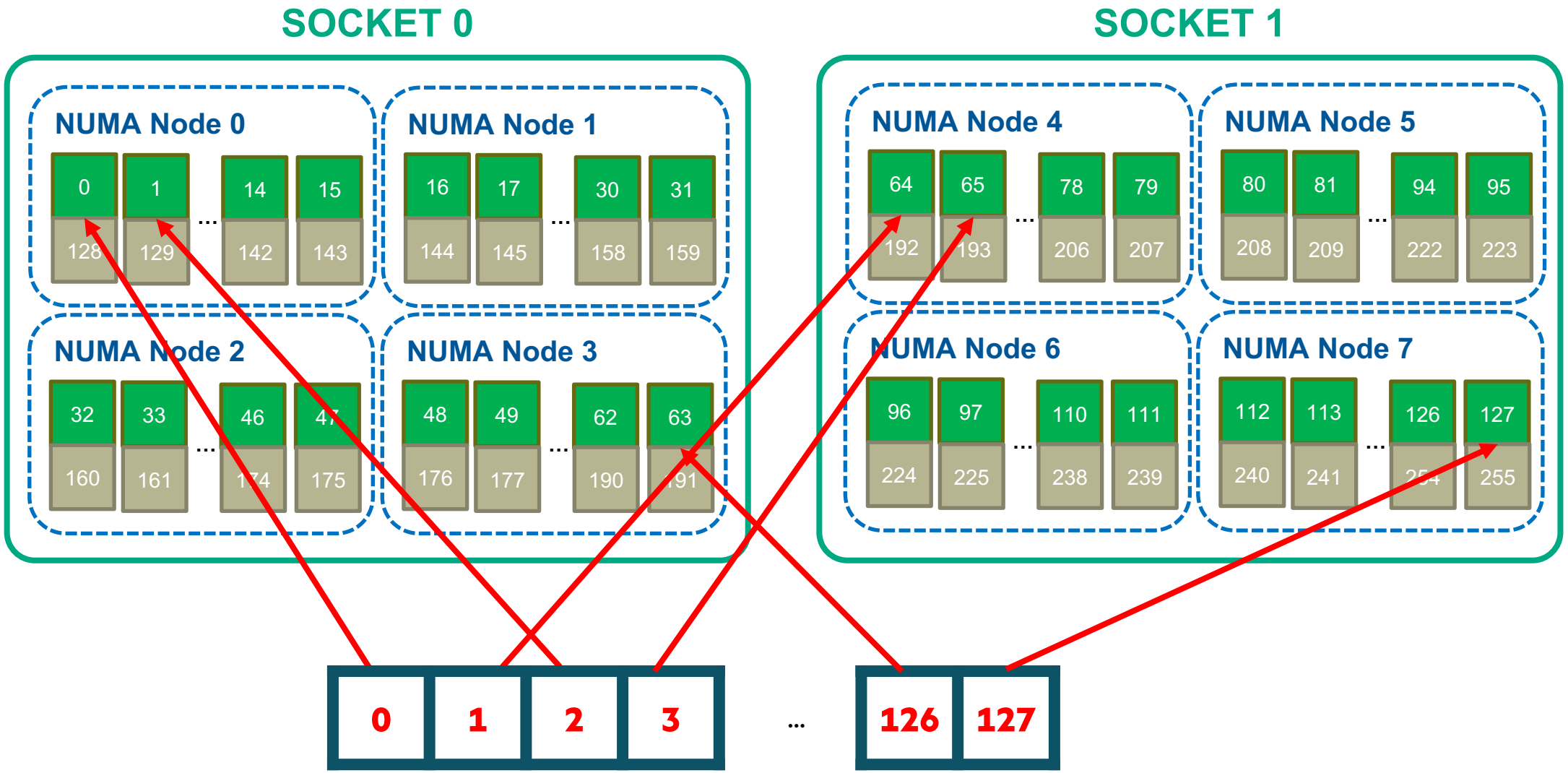


Cyclic : Consecutive tasks are distributed over sockets

--hint=nomultithread

distribution X:CYCLIC

```
> srun -N 1 --ntasks 128 --distribution=block:cyclic ./{EXE}
```



Cyclic : Consecutive tasks are distributed over consecutive NUMA region

LEVEL 3

Threads Binding with OpenMP

Openmp binding

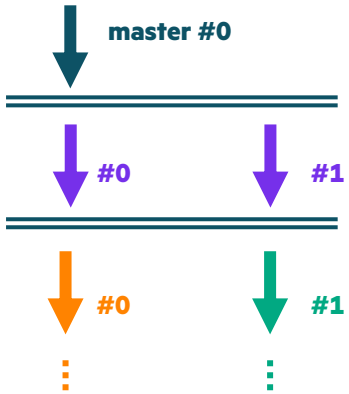
- **From OpenMP v4.0, OpenMP provides `environment` variables to specify how OpenMP threads should be bound to the system hardware.**

- **The variables are**
  - `OMP_NUM_THREADS` - sets the **number of threads** to use for parallel regions
  - `OMP_PLACES` - **specifies places** on the machine where the threads are put.
  - `OMP_PROC_BIND` - **specifies a binding** policy

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

OMP_NUM_THREADS - Set the number of OpenMP threads

```
export OMP_NUM_THREADS=2
```

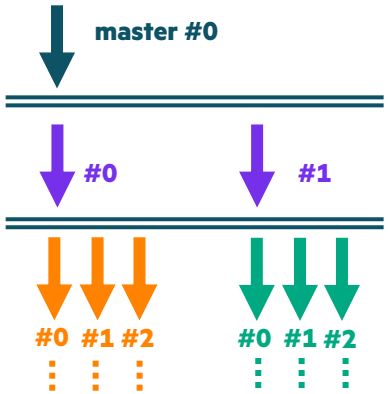


Outer : create 1 new thread to make 1 team of 2 threads

Inner : No new threads created to make 2 teams of 1 threads

```
outer: thread_num = 0 ... level = 1
outer: thread_num = 1 ... level = 1
inner: thread_num = 0 ... level = 2
inner: thread_num = 0 ... level = 2
```

```
export OMP_NUM_THREADS=2,2
```



Outer : create 1 new thread to make 1 team of 2 threads

Inner : 4 new threads to make 2 teams of 3 threads

```
outer: thread_num = 0 ... level = 1
outer: thread_num = 1 ... level = 1
inner: thread_num = 0 ... level = 2
inner: thread_num = 0 ... level = 2
inner: thread_num = 1 ... level = 2
inner: thread_num = 1 ... level = 2
inner: thread_num = 2 ... level = 2
inner: thread_num = 2 ... level = 2
```

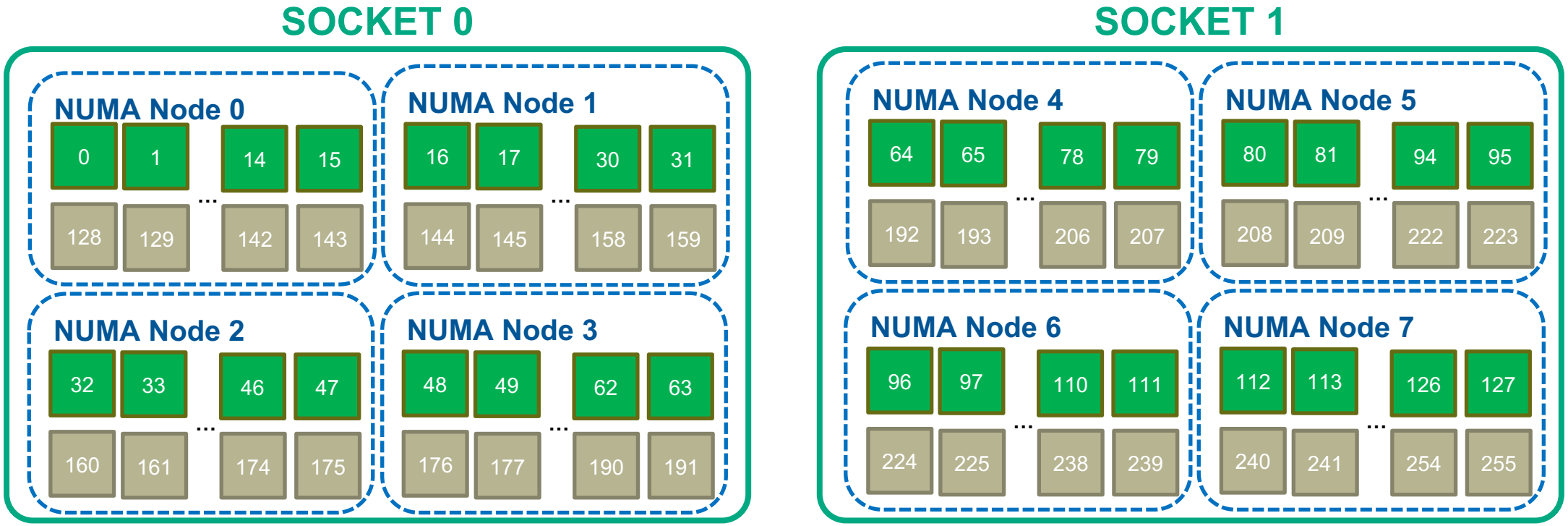
```
int main() {
    #pragma omp parallel
        printf("outer: ...");
        #pragma omp parallel
            printf("inner: ... ");
}
```

Omp_places

- A list of places that threads can be pinned on.
- Each "**Place**" defines a location where a **thread can "float"**
- Places do not depend on the number of threads (OMP_NUM_THREADS)

- OMP_PLACES=*VALUES* where 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)
  - **sockets**: Each place corresponds **to a single socket** (consisting of one or more cores)
  - Or ***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}"

PLACES = THREADS




```
> srun -N 1 --ntasks 8 -c 32
```

OMP_DISPLAY_ENV=true

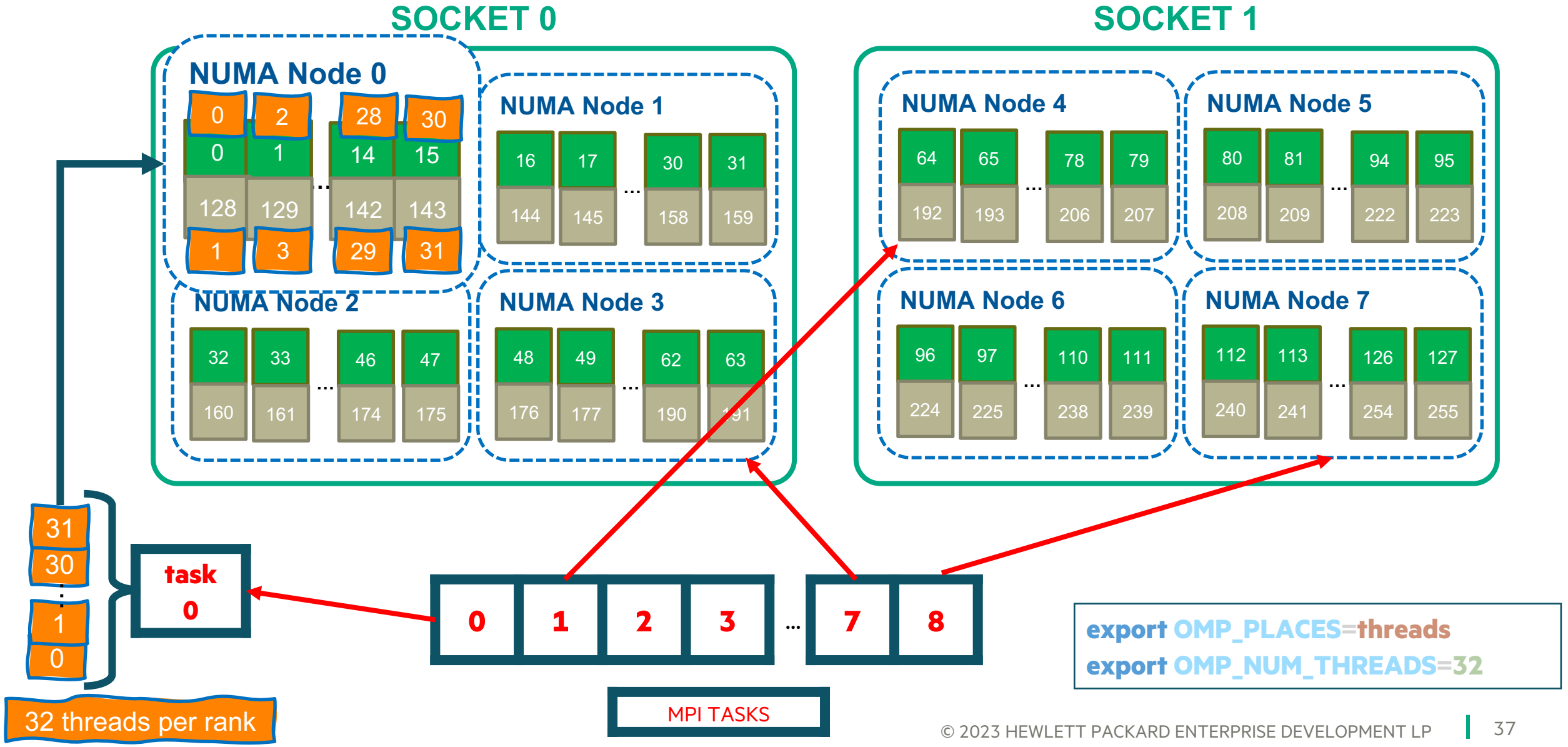
```
OMP_PLACES = '{0},{128},{1},{129},{2},{130},{3} ...
OMP_PLACES = '{16},{144},{17},{145},{18},{146},{19} ...
OMP_PLACES = '{32},{160},{33},{161},{34},{162},{35} ...
OMP_PLACES = '{48},{176},{49},{177},{50},{178},{51} ...
OMP_PLACES = '{64},{192},{65},{193},{66},{194},{67} ...
OMP_PLACES = '{80},{208},{81},{209},{82},{210},{83} ...
OMP_PLACES = '{96},{224},{97},{225},{98},{226},{99} ...
OMP_PLACES = '{112},{240},{113},{241},{114},{242},{115} ...
```

OMP_PLACES=threads

1 place = a single hardware thread

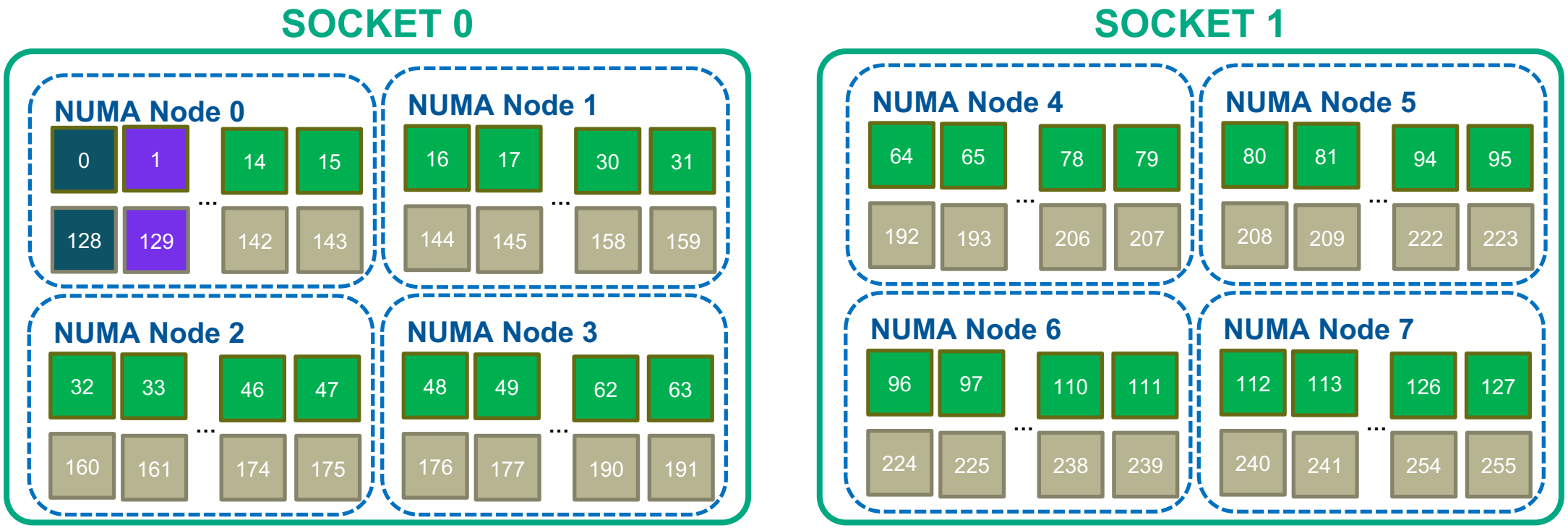
PLACES = THREADS

```
> srun -N 1 --ntasks 8 -c 32 --distribution=block:cyclic ./${EXE}
```




```
export OMP_PLACES=threads
export OMP_NUM_THREADS=32
```

PLACES = CORES



OMP_PLACES=cores

1 place = a single core

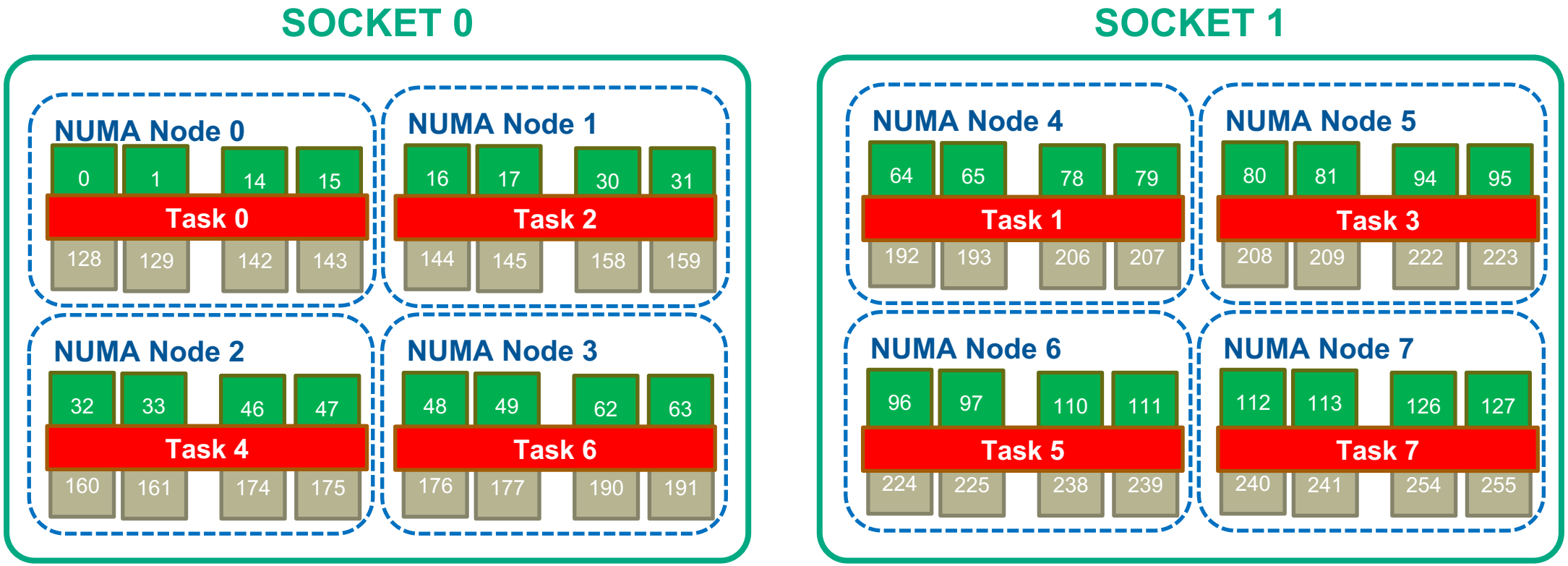
```
> srun -N 1 --ntasks 8 -c 32
```

OMP_DISPLAY_ENV=true

```
OMP_PLACES = '{0,128},{1,129},{2,130},{3,131} ...
OMP_PLACES = '{16,144},{17,145},{18,146},{19,147} ...
OMP_PLACES = '{32,160},{33,161},{34,162},{35,163} ...
OMP_PLACES = '{48,176},{49,177},{50,178},{51,179} ...
OMP_PLACES = '{64,192},{65,193},{66,194},{67,195} ...
OMP_PLACES = '{80,208},{81,209},{82,210},{83,211} ...
OMP_PLACES = '{96,224},{97,225},{98,226},{99,227} ...
OMP_PLACES = '{112,240},{113,241},{114,242},{115,243} ...
```

PLACES = SOCKETS

```
> srun --nodes 1 -n 8 -c 32 --distribution=block:cyclic: --hint=multithread
```




```
OMP_PLACES = '{0:16,128:16}'
OMP_PLACES = '{16:16,144:16}'
OMP_PLACES = '{32:16,160:16}'
OMP_PLACES = '{48:16,176:16}'
OMP_PLACES = '{64:16,192:16}'
OMP_PLACES = '{80:16,208:16}'
OMP_PLACES = '{96:16,224:16}'
OMP_PLACES = '{112:16,240:16}'
```

```
host        rank thr  count   mask
nid002241      0   0   32  cpus 0-15 128-143
               1   0   32  cpus 64-79 192-207
               2   0   32  cpus 16-31 144-159
               3   0   32  cpus 80-95 208-223
               4   0   32  cpus 32-47 160-175
               5   0   32  cpus 96-111 224-239
               6   0   32  cpus 48-63 176-191
               7   0   32  cpus 112-127 240-255
```

```
OMP_PLACES=sockets
OMP_NUM_THREADS=1
```

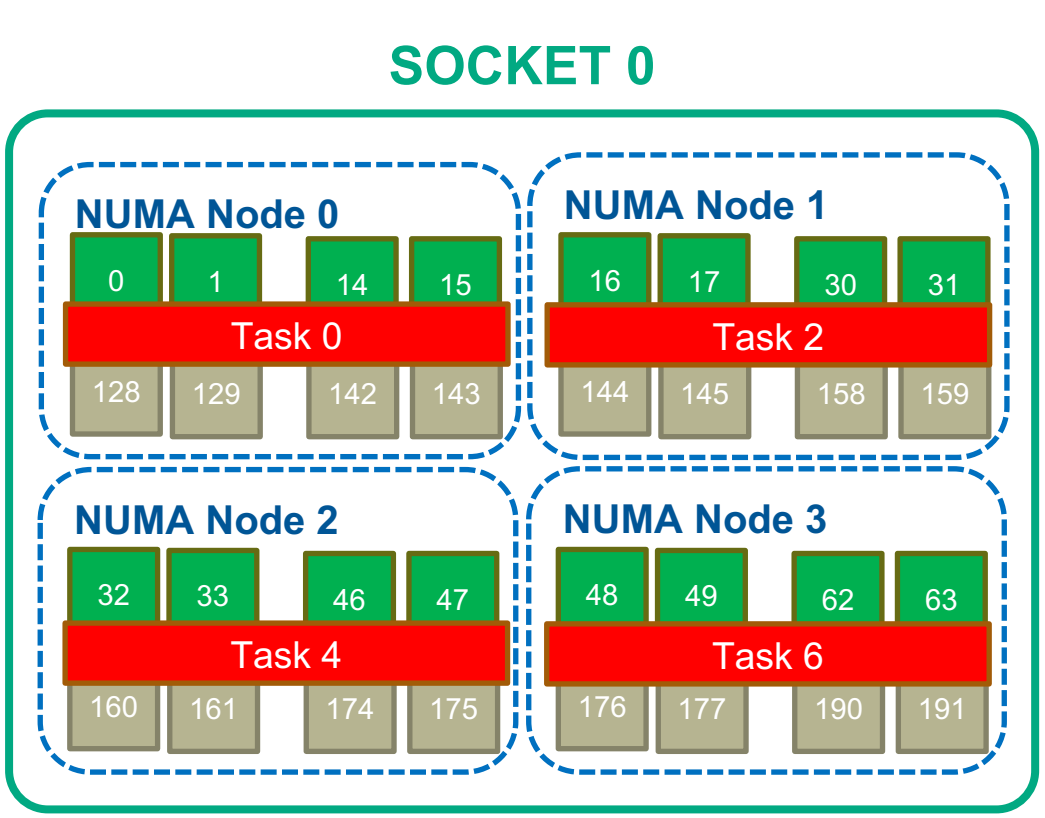
threads can "float"

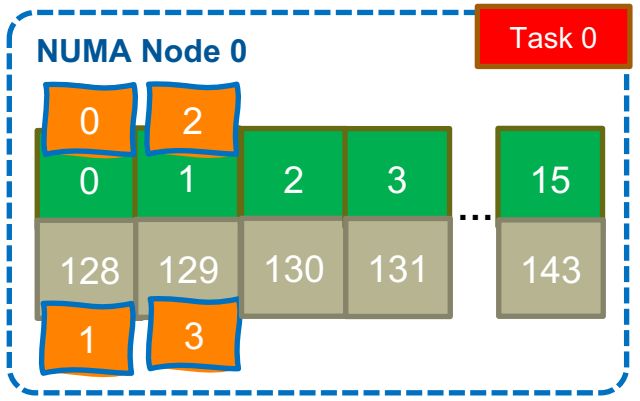
Omp_proc_bind

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

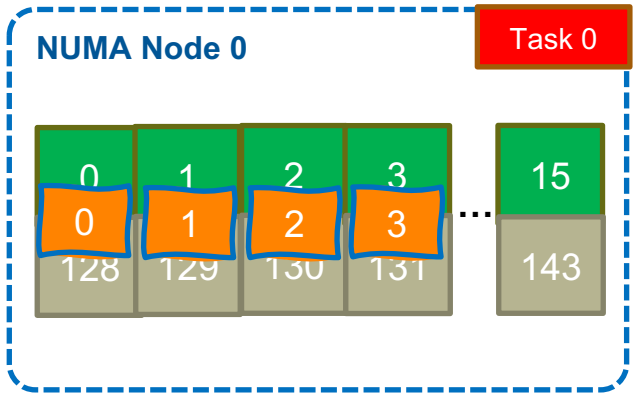
Omp_proc_bind=close

- Here we specify 8 MPI tasks with 4 OpenMP threads with places = threads or cores






```
export OMP_PROC_BIND=close
export OMP_PLACES=threads
export OMP_NUM_THREADS=4

      host rank thr   count mask
 nid002241    0   0   1 cpu 0
                  1   1 cpu 128
                  2   1 cpu 1
                  3   1 cpu 129
              1   0   1 cpu 64
...
```

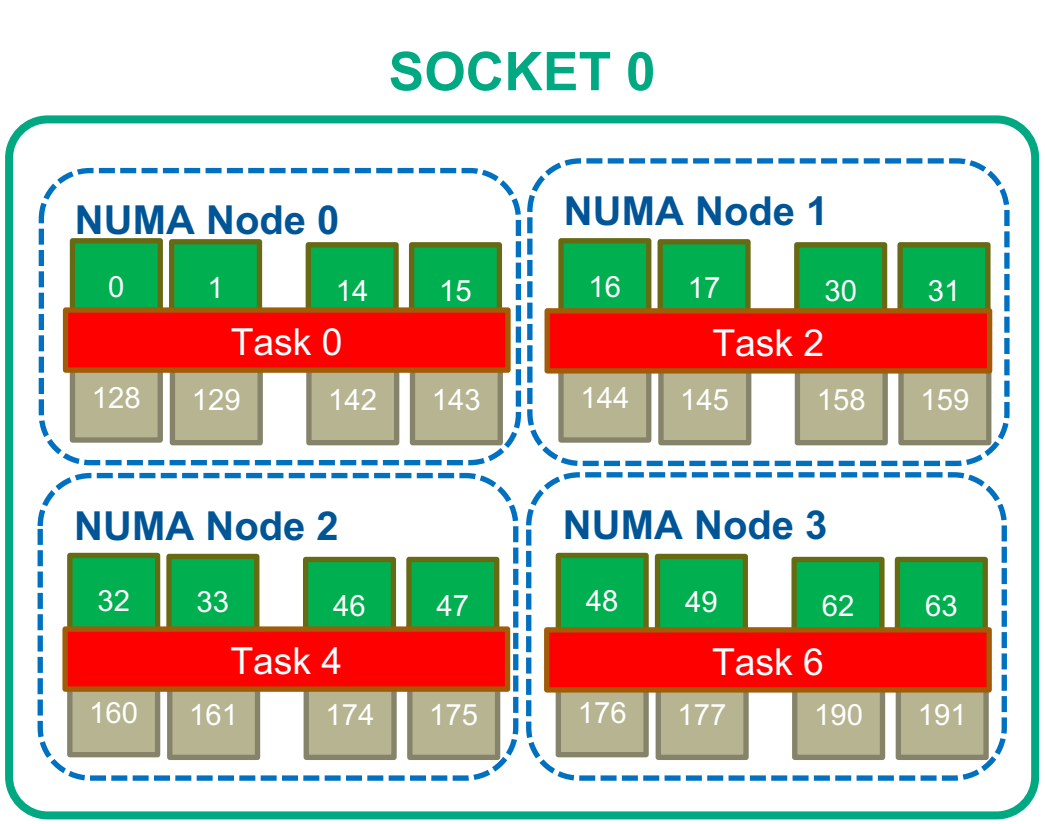


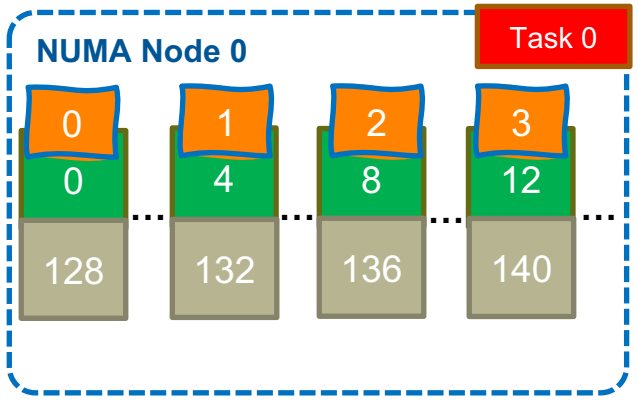

```
export OMP_PROC_BIND=close
export OMP_PLACES=cores
export OMP_NUM_THREADS=4

      host rank thr   count   mask
 nid002241    0   0   2 cpus 0 128
                  1   2 cpus 1 129
                  2   2 cpus 2 130
                  3   2 cpus 3 131
              1   0   2 cpus 64 192
...
```

```
> srun -N 1 --ntasks 8 -c 32--distribution=block:cyclic --hint=multithread ./${EXE}
```

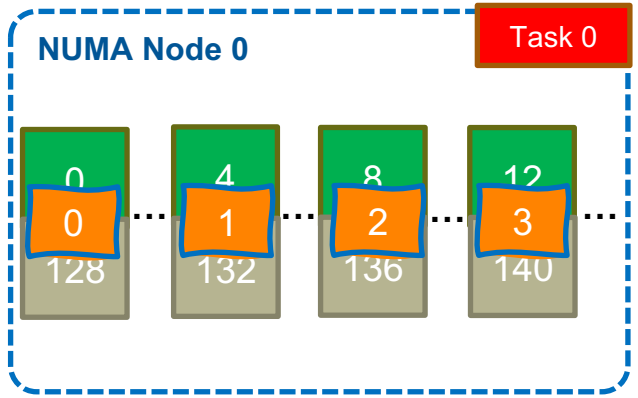
Omp_proc_bind=spread

- Here we specify 8 MPI tasks with 4 OpenMP threads with places = threads or cores






```
export OMP_PROC_BIND=spread
export OMP_PLACES=threads
export OMP_NUM_THREADS=4

      host rank thr   count mask
nid002241    0   0   1 cpu 0
                 1   1 cpu 4
                 2   1 cpu 8
                 3   1 cpu 12
             1   0   1 cpu 64
...
```

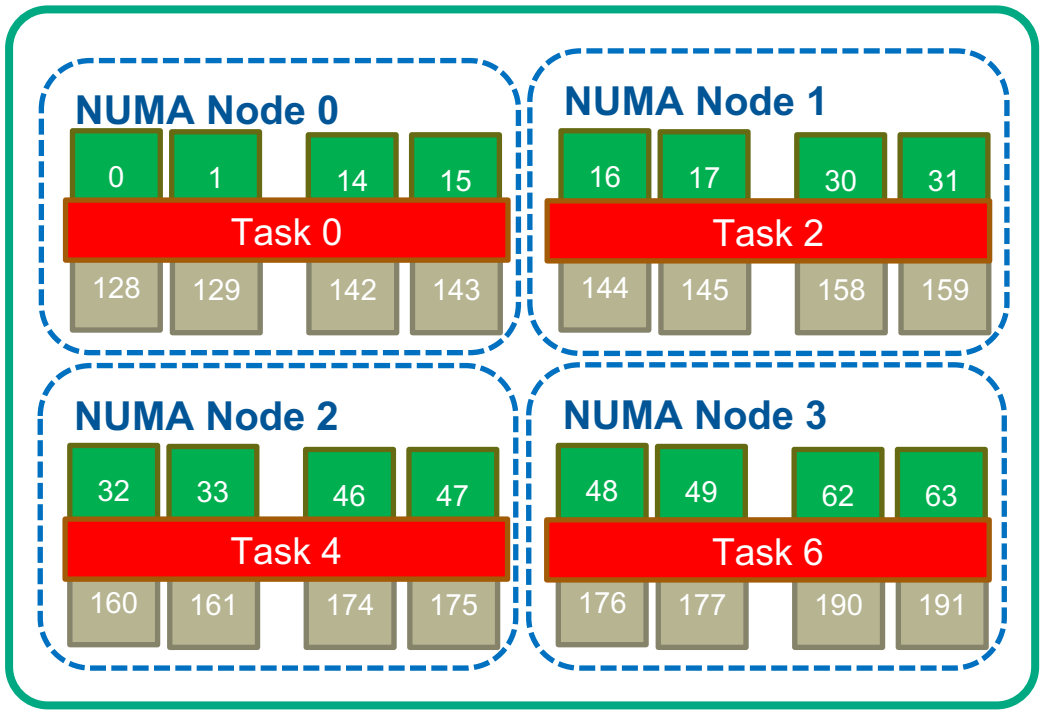


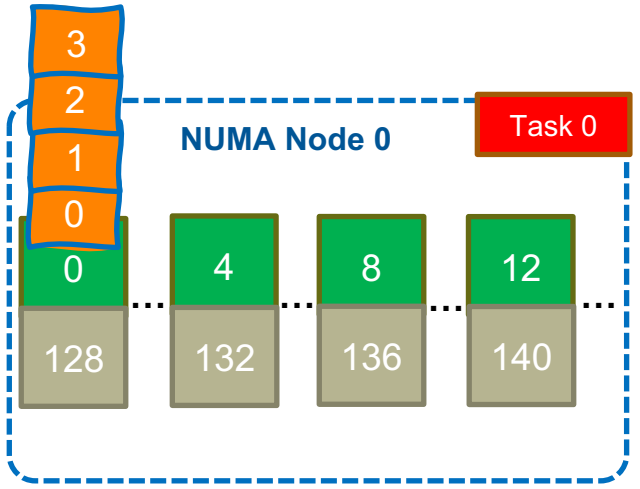

```
export OMP_PROC_BIND=spread
export OMP_PLACES=cores
export OMP_NUM_THREADS=4

      host rank thr   count   mask
nid002241    0   0   2 cpus 0 128
                 1   2 cpus 4 132
                 2   2 cpus 8 136
                 3   2 cpus 12 140
             1   0   2 cpus 64 192
...
```

```
> srun -N 1 --ntasks 8 -c 32--distribution=block:cyclic --hint=multithread ./{EXE}
```

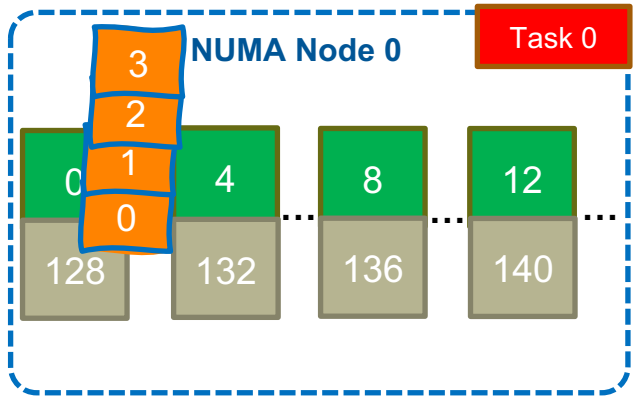
Omp_proc_bind=master

- Here we specify 8 MPI tasks with 4 OpenMP threads with places = threads or cores






```
export OMP_PROC_BIND=master
export OMP_PLACES=threads
export OMP_NUM_THREADS=4

      host  rank  thr   count  mask
 nid002241     0    0   1 cpu  0
                    1   1 cpu  0
                    2   1 cpu  0
                    3   1 cpu  0
                1   0   1 cpu  64
...
```

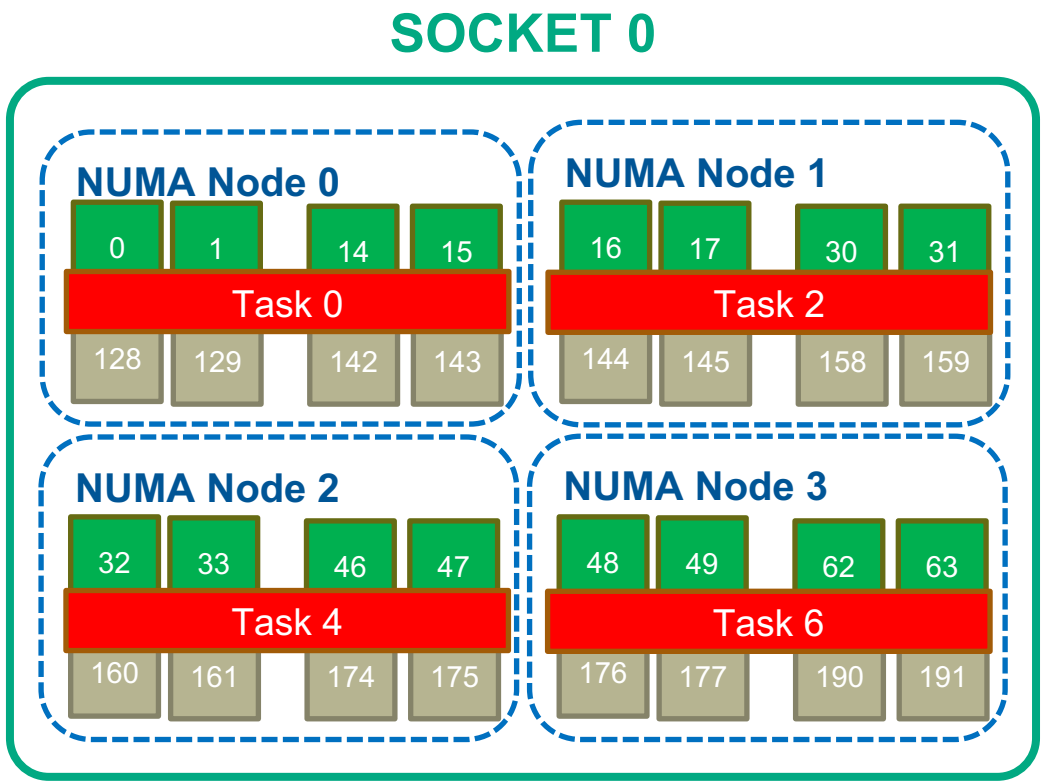


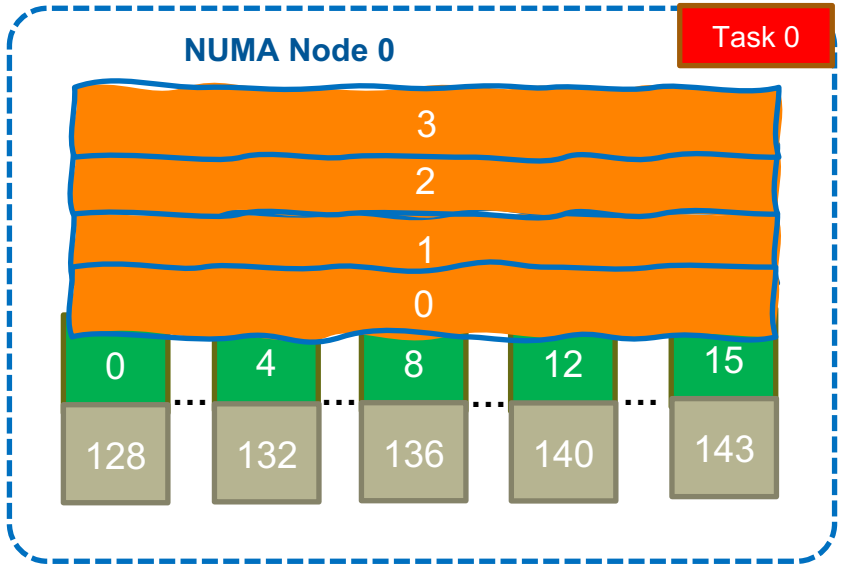

```
export OMP_PROC_BIND=master
export OMP_PLACES=cores
export OMP_NUM_THREADS=4

      host  rank  thr   count   mask
 nid002241     0    0   2 cpus  0 128
                    1   2 cpus  0 128
                    2   2 cpus  0 128
                    3   2 cpus  0 128
                1   0   2 cpus  64 192
...
```

```
> srun -N 1 --ntasks 8 -c 32--distribution=block:cyclic --hint=multithread ./{EXE}
```

Omp_proc_bind=false

- Here we specify 8 MPI tasks with 4 OpenMP threads with places = threads or cores






```
export OMP_PROC_BIND=false
export OMP_PLACES=threads | cores | sockets #same results
export OMP_NUM_THREADS=4
 host rank thr   count    mask
 nid002241    0    0   32 cpus 0-15 128-143
                   1   32 cpus 0-15 128-143
                   2   32 cpus 0-15 128-143
                   3   32 cpus 0-15 128-143
              1    0   32 cpus 64-79 192-207
```

```
> srun -N 1 --ntasks 8 -c 32--distribution=block:cyclic --hint=multithread ./{EXE}
```

SLURM OPTIONS

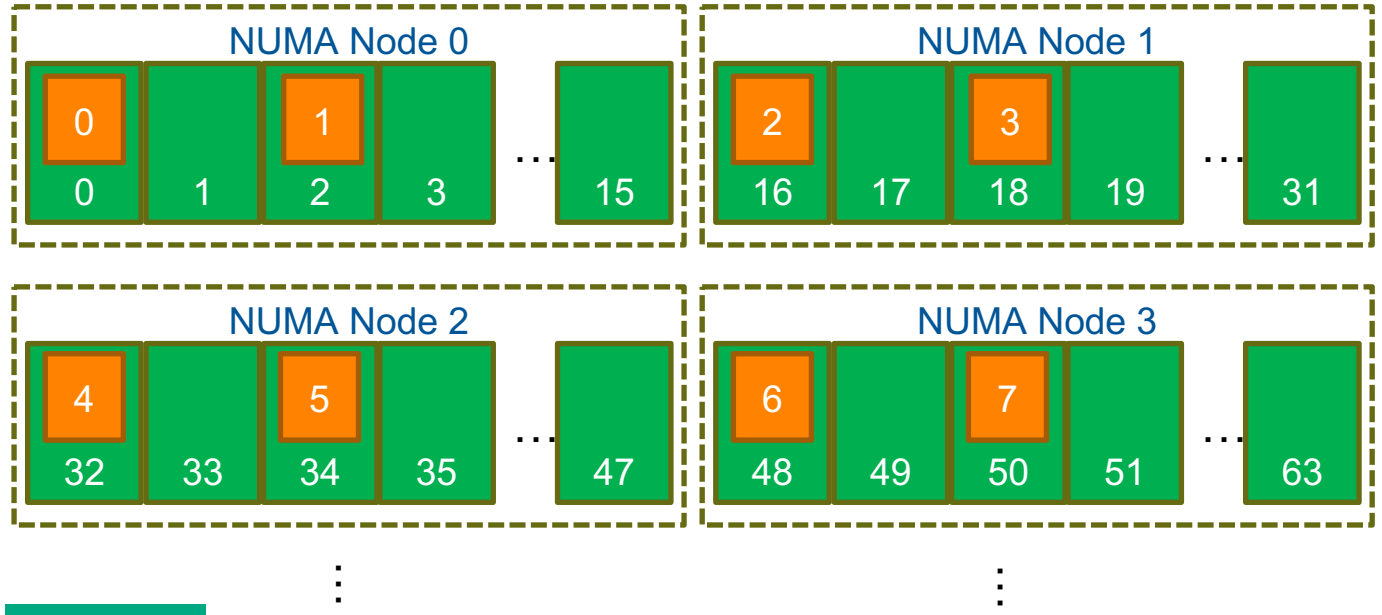
Other distribution techniques

Custom Binding with a Map

- The user can bind tasks explicitly to specific CPUs using a listed map
- The mapping is the same for all nodes
- Useful when underpopulating a node to access more memory bandwidth per task

```
export bind=0,2,16,18,32,34,48,50
srun -n 1 -n 8 --cpu-bind=map_cpu:${bind} ./${EXE}
```

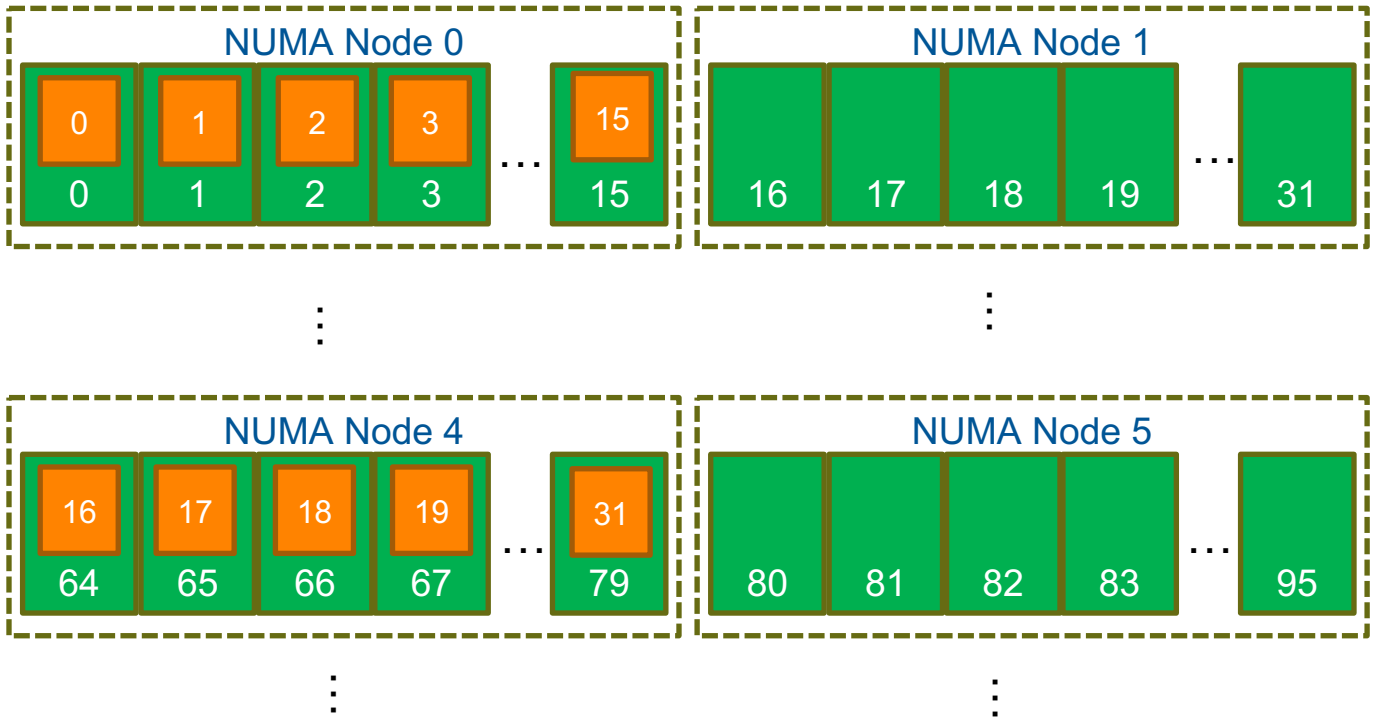
```
      host rank thr  count mask
      nid002459    0   0  1 cpu 0
                   1   0  1 cpu 2
                   2   0  1 cpu 16
                   3   0  1 cpu 18
                   4   0  1 cpu 32
                   5   0  1 cpu 34
                   6   0  1 cpu 48
                   7   0  1 cpu 50
```



Specifying a Number of Tasks per Socket

- The number of tasks per socket can be limited

```
srun --nodes 1 -n 32 --ntasks-per-socket=16 --hint=nomultithread ./${EXE}
```



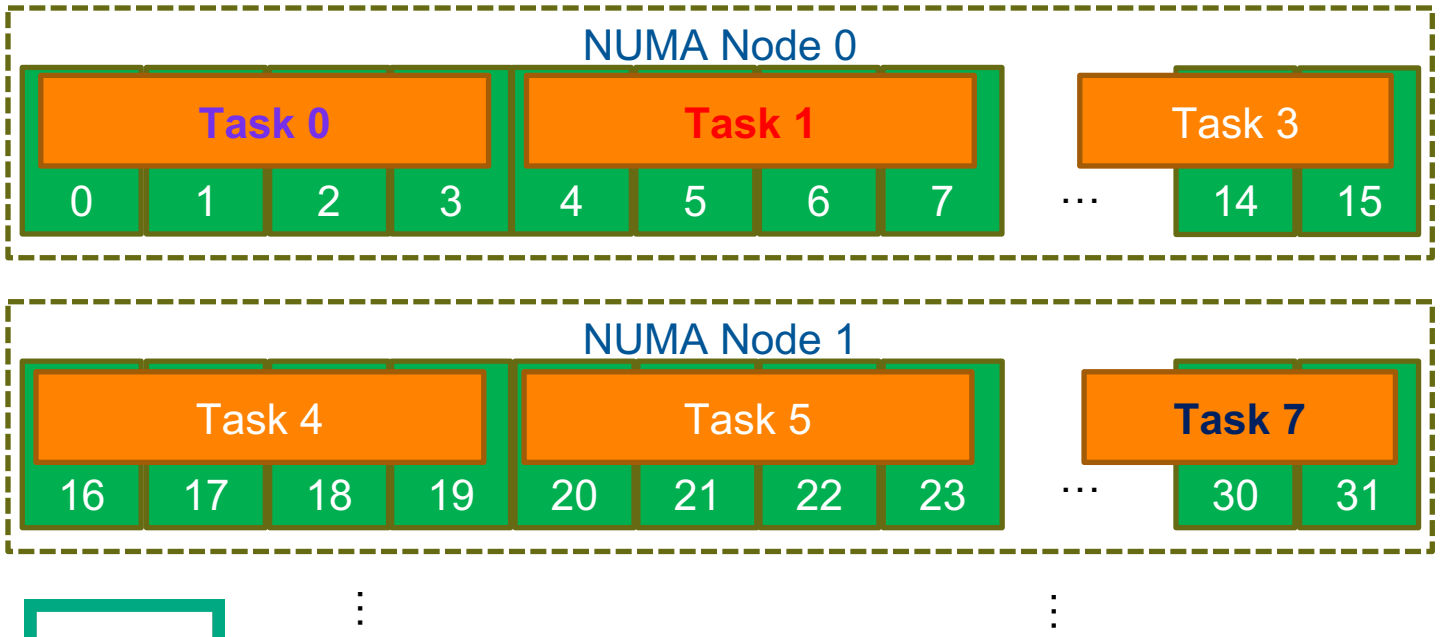

```
host rank thr  count mask
 nid002459    0    0  1 cpu 0
              1    0  1 cpu 1
...
             14    0  1 cpu 14
             15    0  1 cpu 15
             16    0  1 cpu 64
             17    0  1 cpu 65
...
             30    0  1 cpu 78
             31    0  1 cpu 79
```

srun will not inherit the --cpus-per-task value requested by salloc or sbatch

Hybrid Binding

- To define the number of threads per task use **--cpus-per-task (-c)**
- Make sure that **#threads** divides the **#cores** on a socket
- This can be fixed by forcing the task to another socket

```
export OMP_NUM_THREADS=4
export OMP_PLACES=sockets
srun -n8 --cpus-per-task 4 \\
         --distribution=block:block ./${EXE}
```




```
nid002459     0   0  4 cpus 0-3
                  1  4 cpus 0-3
                  2  4 cpus 0-3
                  3  4 cpus 0-3
              1   0  4 cpus 4-7
                  1  4 cpus 4-7
                  2  4 cpus 4-7
                  3  4 cpus 4-7
...
              6   0  4 cpus 24-27
                  1  4 cpus 24-27
                  2  4 cpus 24-27
                  3  4 cpus 24-27
              7   0  4 cpus 28-31
                  1  4 cpus 28-31
                  2  4 cpus 28-31
                  3  4 cpus 28-31
```

Final Remarks

CPU Binding Summary

- Experiment with binding options in your applications
  - Can improve performance depending on the used MPI-communication pattern

- Use xthi-like applications to check the affinity
  - You can have it in your submission script before running your application, e.g.

```
srun ./xthi | sort -n -k 4 -k 6
srun <my_app>
```

- Although a bit difficult, it should only be done once
- Use reporting bindings
  - Slurm **`--cpu-bind=verbose,<mode>`**
  - **`OMP_DISPLAY_AFFINITY=true`**
  - **`MPICH_CPUMASK_DISPLAY=1`**

Task Binding for GPU

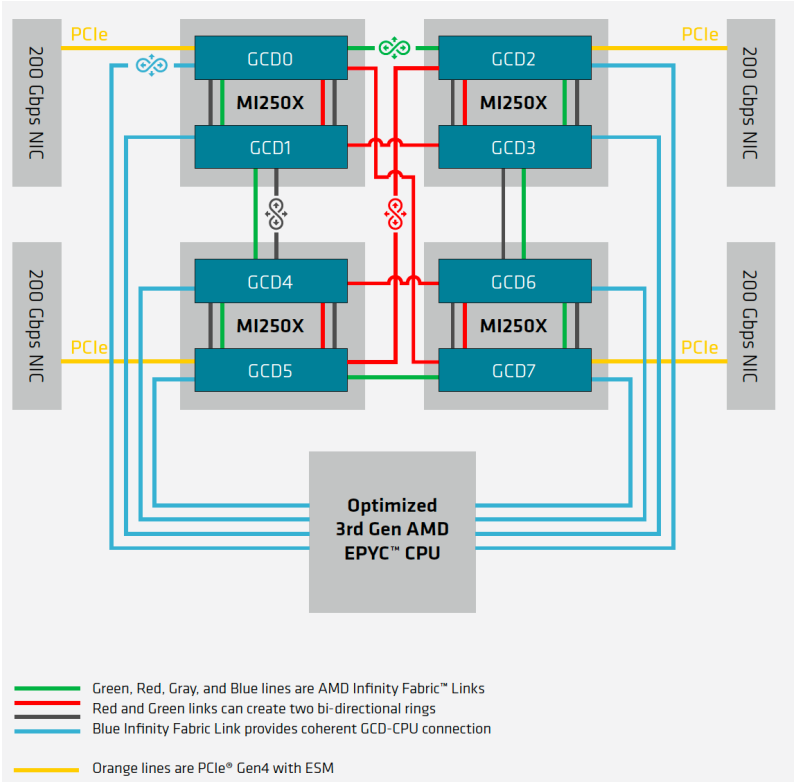
Viewing Available Resources

On **Lumi-G**:

- 4 MI250X per node = 8 GCDs in total
- Will show as 8 separate GPUs according to Slurm,
  - `HIP_VISIBLE_DEVICES`, `ROCR_VISIBLE_DEVICES`

Note :

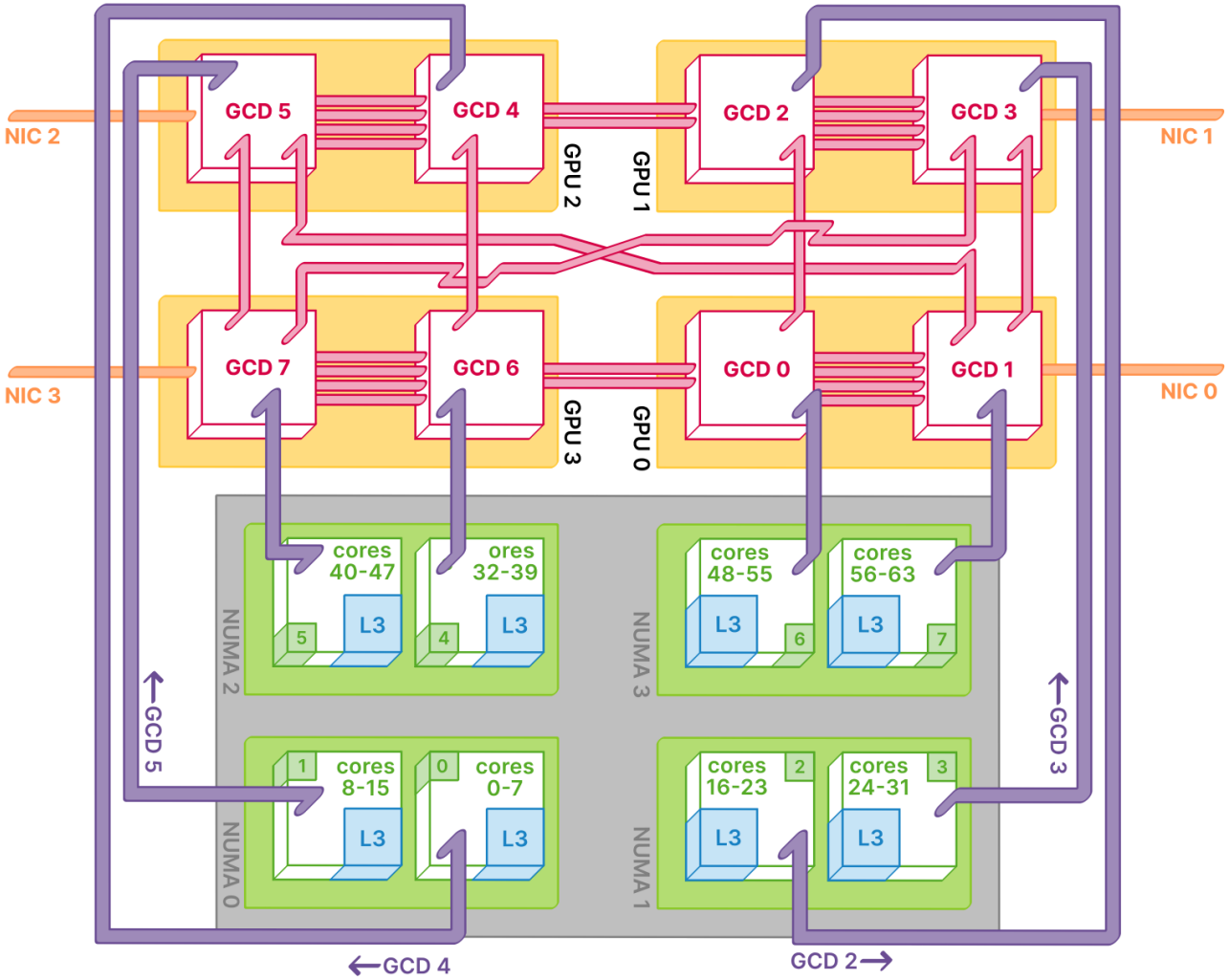
- We will refer to the GCDs as GPUs
- Keep in mind that 2 GDCs can be on the same physical package
- This can (will) affect performance

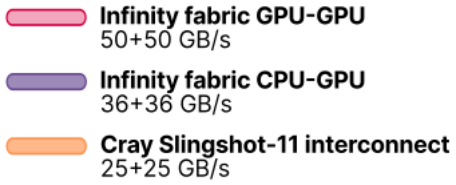


One MI250X GPU is spitted in **Two GCDs**

```
uan02:~> sinfo -o "%10R  %10c  %10m  %20G  %20F" -p main,gpu
PARTITION   CPUS        MEMORY      GRES                 NODES(A/I/O/T)
main        256         212992+     (null)               715/239/46/1000
gpu         128         456704      (null)               9/46/1/56
```

AMD CPU/GPU TOPOLOGY

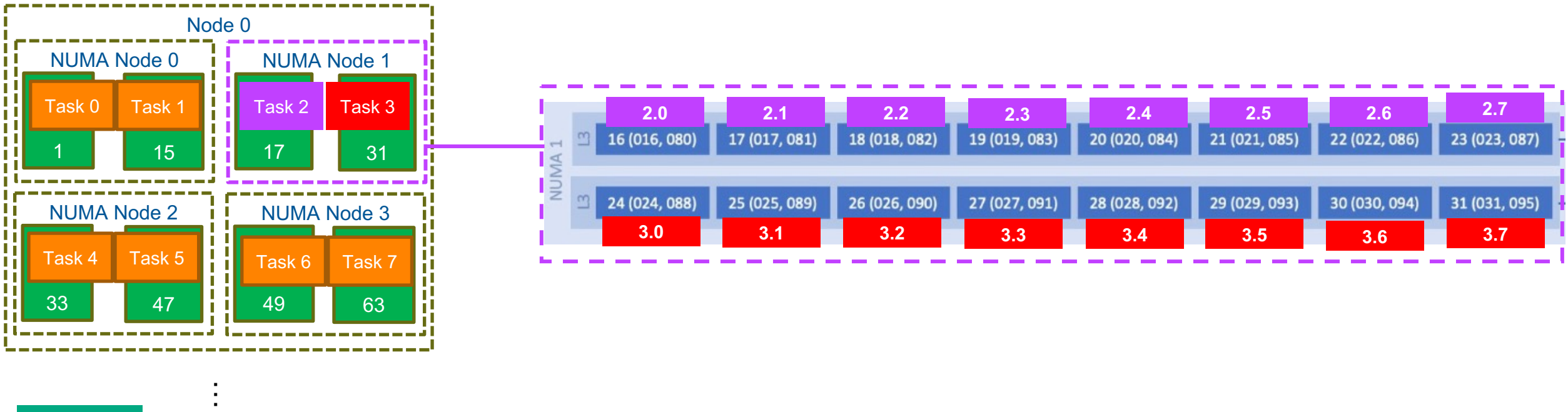




source : https://docs.lumi-supercomputer.eu/hardware/lumig/

Combining MPI Tasks and OpenMP Threads (2)

- 8 tasks/node
- No SMT threads
- 8 threads/rank
- “Compact” binding



COMBINING MPI TASKS AND OPENMP THREADS: DEFAULT EXAMPLE

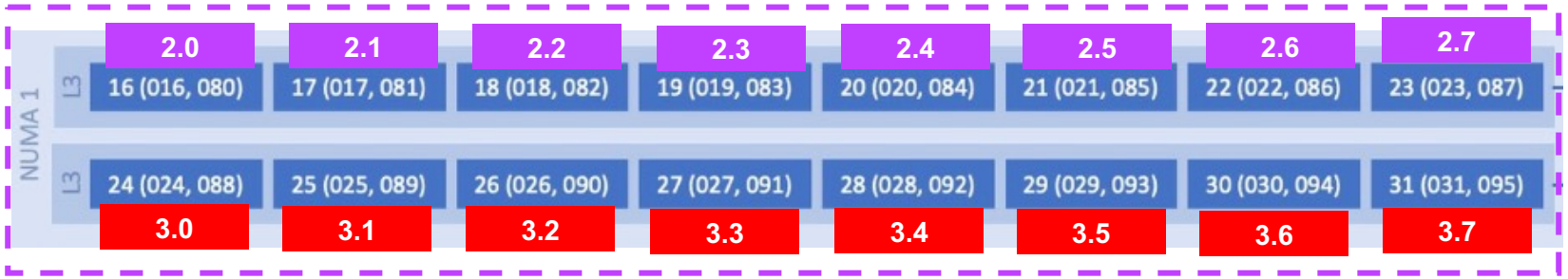
- 8 tasks/node
- No SMT threads
- 8 threads/rank
- “Compact” binding

- Submission script
  - xthi example

```
#!/bin/bash
#SBATCH --exclusive
#SBATCH --time=00:02:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=8
#SBATCH --cpus-per-task=8
#SBATCH --hint=nomultithread


export OMP_PLACES=cores
export OMP_PROC_BIND=close
export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASK}

srun --cpus-per-task=8 xthi | sort -n -k 4 -k 6
```

COMBINING MPI TASKS AND OPENMP THREADS: DEFAULT EXAMPLE

What we want :



Xthi output :

```
Hello from rank 2, thread 0, on nid002814. (core affinity = 16)
Hello from rank 2, thread 1, on nid002814. (core affinity = 17)
Hello from rank 2, thread 2, on nid002814. (core affinity = 18)
Hello from rank 2, thread 3, on nid002814. (core affinity = 19)
Hello from rank 2, thread 4, on nid002814. (core affinity = 20)
Hello from rank 2, thread 5, on nid002814. (core affinity = 21)
Hello from rank 2, thread 6, on nid002814. (core affinity = 22)
Hello from rank 2, thread 7, on nid002814. (core affinity = 23)
Hello from rank 3, thread 0, on nid002814. (core affinity = 24)
Hello from rank 3, thread 1, on nid002814. (core affinity = 25)
Hello from rank 3, thread 2, on nid002814. (core affinity = 26)
Hello from rank 3, thread 3, on nid002814. (core affinity = 27)
Hello from rank 3, thread 4, on nid002814. (core affinity = 28)
Hello from rank 3, thread 5, on nid002814. (core affinity = 29)
Hello from rank 3, thread 6, on nid002814. (core affinity = 30)
Hello from rank 3, thread 7, on nid002814. (core affinity = 31)
Hello from rank 4, thread 0, on nid002814. (core affinity = 32)
Hello from rank 4, thread 1, on nid002814. (core affinity = 33)
```

GPU AFFINITY

CPU / GPU AFFINITY

MPI Ranks = 8

- Pinned to the 8 CCD

OpenMP Threads = 8

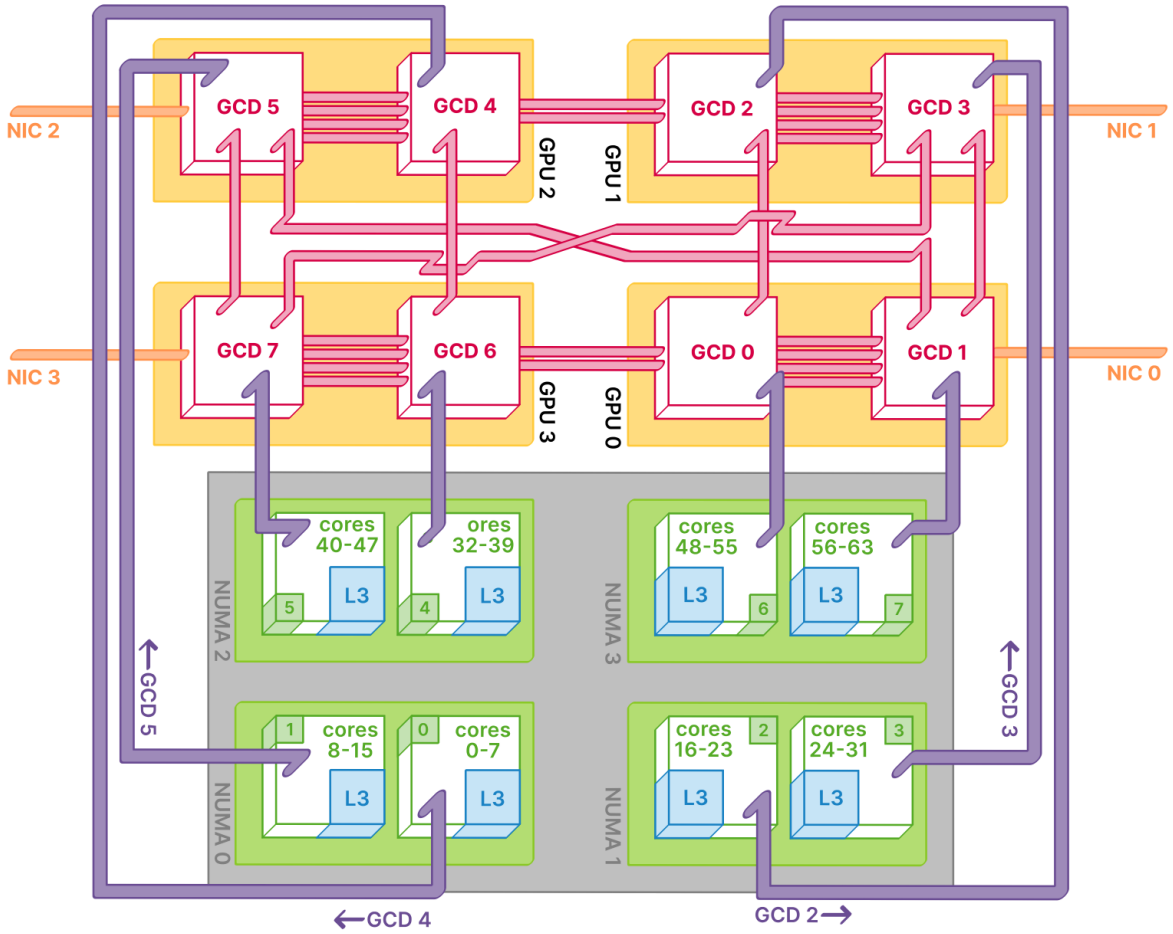
Credit : ORNL

GPUS TO NUMA DOMAINS MAPPING

- GPUs are associated to NUMA nodes
- Can use rocm-smi to see the topology

```
> srun --nodes=1 -t "00:10:00" --ntasks=1 --gres=gpu:8 rocm-smi –showtopo | grep Affinity
…
GPU[0]          : (Topology) Numa Affinity: 3
GPU[1]          : (Topology) Numa Affinity: 3
GPU[2]          : (Topology) Numa Affinity: 1
GPU[3]          : (Topology) Numa Affinity: 1
GPU[4]          : (Topology) Numa Affinity: 0
GPU[5]          : (Topology) Numa Affinity: 0
GPU[6]          : (Topology) Numa Affinity: 2
GPU[7]          : (Topology) Numa Affinity: 2
```

GPUS TO NUMA DOMAINS MAPPING



NUMA ID	0	0	1	1	2	2	3	3
Optimal GPU ID	4	5	2	3	6	7	0	1

GPUS BINDING - ASSUMING ONE GPU PER MPI RANK

- Associate each MPI rank to a given GPU with two environment variables:
  - **`ROCR_VISIBLE_DEVICES`**
    - Limit the number of GPU devices that are available for a given process
  - **`SLURM_LOCALID`**
    - Node local task ID for the process within a job

- You can check GPU bindings via the tool available at https://code.ornl.gov/olcf/hello_jobstep

```
MPI 000 - OMP 000 - HWT 001 - Node nid005015 - RT_GPU_ID 0,1,2,3,4,5,6,7 - GPU_ID 0,1,2,3,4,5,6,7 - Bus_ID c1,c6,c9,ce,d1,d6,d9,de
MPI 000 - OMP 001 - HWT 002 - Node nid005015 - RT_GPU_ID 0,1,2,3,4,5,6,7 - GPU_ID 0,1,2,3,4,5,6,7 - Bus_ID c1,c6,c9,ce,d1,d6,d9,de
MPI 000 - OMP 002 - HWT 003 - Node nid005015 - RT_GPU_ID 0,1,2,3,4,5,6,7 - GPU_ID 0,1,2,3,4,5,6,7 - Bus_ID c1,c6,c9,ce,d1,d6,d9,de
MPI 000 - OMP 003 - HWT 004 - Node nid005015 - RT_GPU_ID 0,1,2,3,4,5,6,7 - GPU_ID 0,1,2,3,4,5,6,7 - Bus_ID c1,c6,c9,ce,d1,d6,d9,de
…
MPI 007 - OMP 004 - HWT 060 - Node nid005015 - RT_GPU_ID 0,1,2,3,4,5,6,7 - GPU_ID 0,1,2,3,4,5,6,7 - Bus_ID c1,c6,c9,ce,d1,d6,d9,de
MPI 007 - OMP 005 - HWT 061 - Node nid005015 - RT_GPU_ID 0,1,2,3,4,5,6,7 - GPU_ID 0,1,2,3,4,5,6,7 - Bus_ID c1,c6,c9,ce,d1,d6,d9,de
MPI 007 - OMP 006 - HWT 062 - Node nid005015 - RT_GPU_ID 0,1,2,3,4,5,6,7 - GPU_ID 0,1,2,3,4,5,6,7 - Bus_ID c1,c6,c9,ce,d1,d6,d9,de
MPI 007 - OMP 007 - HWT 063 - Node nid005015 - RT_GPU_ID 0,1,2,3,4,5,6,7 - GPU_ID 0,1,2,3,4,5,6,7 - Bus_ID c1,c6,c9,ce,d1,d6,d9,de
```

GPUS BINDING – NAIVE APPROACH (1)

- Let's consider the example of the 8 MPI tasks/node shown before
- Use a script to properly set ROCR_VISIBLE_DEVICES during the job submission

job.slurm

```
#!/bin/bash
#SBATCH -p <partition>
#SBATCH -A <your_project>
#SBATCH --time=00:02:00
#SBATCH --nodes=2
#SBATCH --exclusive
#SBATCH --ntasks-per-node=8
#SBATCH --cpus-per-task=8
#SBATCH --hint=nomultithread
...
${ASRUN} ./select_gpu.sh <my_app>
```

select_gpu.sh

```
#!/bin/bash

export ROCR_VISIBLE_DEVICES=$SLURM_LOCALID

exec $*
```

- Naïve approach example, Each MPI rank :
  - Detects the availability of a single GPU device
  - Is essentially associated with a different GPU device to ensure that two MPI ranks do not share the same GPU device

GPUS BINDING – NAIVE APPROACH (2)

- What we want

Local Task ID	0	1	2	3	4	5	6	7
NUMA ID	0	0	1	1	2	2	3	3
Optimal GPU ID	4	5	2	3	6	7	0	1

- Naïve approach binding map

Local Task ID	0	1	2	3	4	5	6	7
Cores ID	0 - 7	8 - 15	16 - 23	24 - 31	33 - 39	41 - 47	49 - 55	57 - 63
NUMA ID	0	0	1	1	2	2	3	3
GPU ID	0	1	2	3	4	5	6	7

- This mapping is not optimal

GPUS BINDING – AFFINITY SCRIPT

- Adapt `select_gpu.sh` script for the optimal GPU ID

```
#!/bin/bash
GPUSID="4 5 2 3 6 7 0 1"
GPUSID=(${GPUSID})
if [ ${#GPUSID[@]} -gt 0 -a -n "${SLURM_NTASKS_PER_NODE}" ]; then
   if [ ${#GPUSID[@]} -gt $SLURM_NTASKS_PER_NODE ]; then
        export ROCR_VISIBLE_DEVICES=${GPUSID[$(($SLURM_LOCALID))]}
    else
        export ROCR_VISIBLE_DEVICES=${GPUSID[$(($SLURM_LOCALID / ($SLURM_NTASKS_PER_NODE / ${#GPUSID[@]}))))]}
    fi
fi
exec $*
```

```
MPI 000 - OMP 000 - GPU_ID 4 - Bus_ID d1
MPI 001 - OMP 000 - GPU_ID 5 - Bus_ID d6
MPI 002 - OMP 000 - GPU_ID 2 - Bus_ID c9
MPI 003 - OMP 000 - GPU_ID 3 - Bus_ID ce
MPI 004 - OMP 000 - GPU_ID 6 - Bus_ID d9
MPI 005 - OMP 000 - GPU_ID 7 - Bus_ID de
MPI 006 - OMP 000 - GPU_ID 0 - Bus_ID c1
MPI 007 - OMP 000 - GPU_ID 1 - Bus_ID c6
```

Local Task ID	0	1	2	3	4	5	6	7
NUMA ID	0	0	1	1	2	2	3	3
Optimal GPU ID	4	5	2	3	6	7	0	1

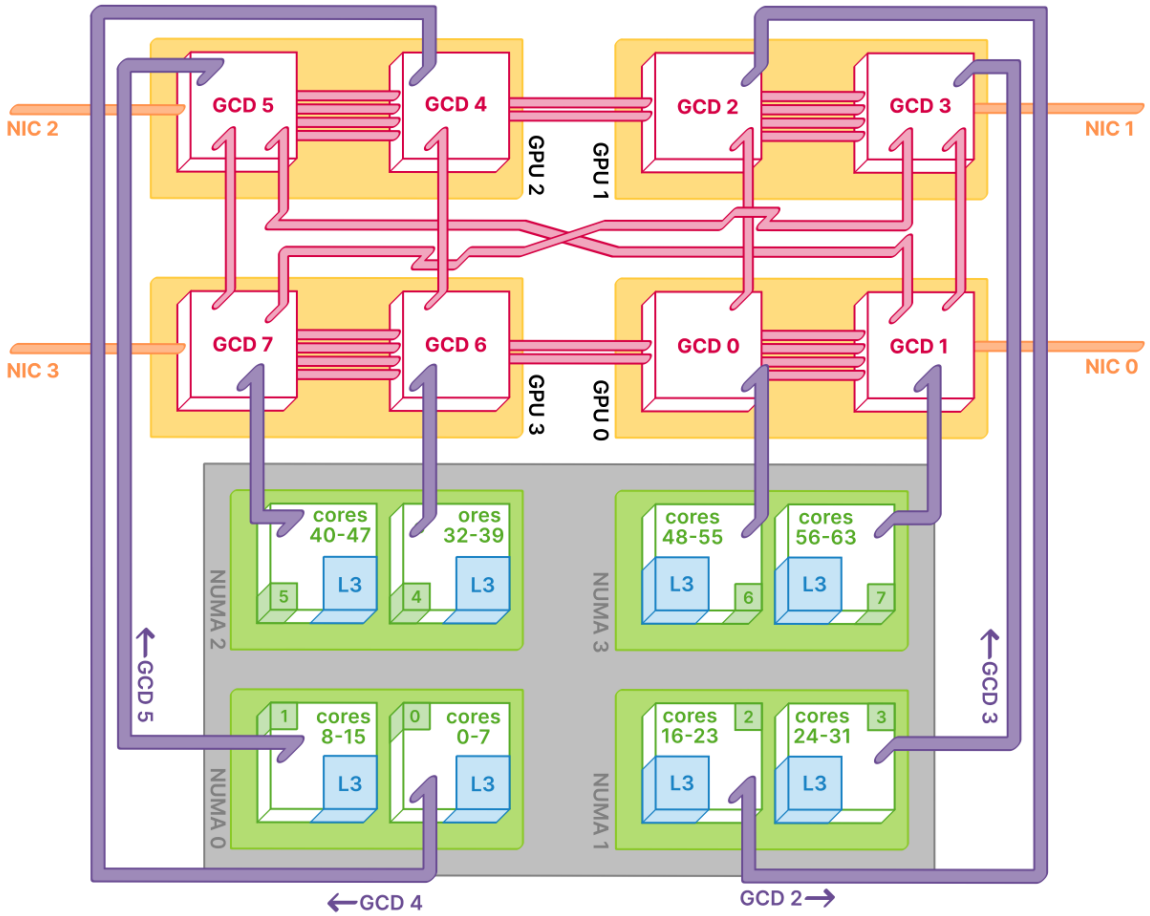
COMBINING MPI TASKS AND OPENMP

- 8 tasks/node
- 8 threads/task
- 8 GPU/node
- 1 GPU/task

```
#!/bin/bash
#SBATCH --nodes=2
#SBATCH --gres=gpu:8
#SBATCH --ntasks-per-node=8
#SBATCH --cpus-per-task=8
#SBATCH --hint=nomultithread
#SBATCH --exclusive

export OMP_PLACES=cores #replace this by sockets
export OMP_PROC_BIND=close
export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASK}

srun ${cpu_bind} ./select_gpu.sh gpu_check -l
```

GPU_CHECK OUTPUT

```
MPI 000 - OMP 000 - HWT 000 (CCD0) GPU_ID 4 - Bus_ID d1(GCD4/CCD0)
MPI 000 - OMP 001 - HWT 001 (CCD0) GPU_ID 4 - Bus_ID d1(GCD4/CCD0)
MPI 000 - OMP 002 - HWT 002 (CCD0) GPU_ID 4 - Bus_ID d1(GCD4/CCD0)
MPI 000 - OMP 003 - HWT 003 (CCD0) GPU_ID 4 - Bus_ID d1(GCD4/CCD0)
MPI 000 - OMP 004 - HWT 004 (CCD0) GPU_ID 4 - Bus_ID d1(GCD4/CCD0)
MPI 000 - OMP 005 - HWT 005 (CCD0) GPU_ID 4 - Bus_ID d1(GCD4/CCD0)
MPI 000 - OMP 006 - HWT 006 (CCD0) GPU_ID 4 - Bus_ID d1(GCD4/CCD0)
MPI 000 - OMP 006 - HWT 007 (CCD0) GPU_ID 4 - Bus_ID d1(GCD4/CCD0)
MPI 001 - OMP 000 - HWT 008 (CCD1) GPU_ID 5 - Bus_ID d6(GCD5/CCD1)
MPI 001 - OMP 001 - HWT 009 (CCD1) GPU_ID 5 - Bus_ID d6(GCD5/CCD1)
MPI 001 - OMP 002 - HWT 010 (CCD1) GPU_ID 5 - Bus_ID d6(GCD5/CCD1)
MPI 001 - OMP 003 - HWT 011 (CCD1) GPU_ID 5 - Bus_ID d6(GCD5/CCD1)
MPI 001 - OMP 004 - HWT 012 (CCD1) GPU_ID 5 - Bus_ID d6(GCD5/CCD1)
MPI 001 - OMP 005 - HWT 013 (CCD1) GPU_ID 5 - Bus_ID d6(GCD5/CCD1)
MPI 001 - OMP 006 - HWT 014 (CCD1) GPU_ID 5 - Bus_ID d6(GCD5/CCD1)
...
```



Infinity fabric GPU-GPU
50+50 GB/s

Infinity fabric CPU-GPU
36+36 GB/s

Cray Slingshot-11 interconnect
25+25 GB/s

MAPPING PROCESSES TO NETWORK INTERFACES

- On compute nodes that offer multiple GPU devices and multiple Network Interface Controllers (NIC), Cray MPI offers a flexible way to offer the ideal mapping between a process and the default NIC
  - We have a NIC per each Mi250X, i.e. 4 NICs per node

- For GPU-enabled parallel applications that involve MPI operations that access application arrays resident are on GPU-attached memory regions, users can set `MPICH_OFI_NIC_POLICY` to "GPU"
  - In this case, for each MPI process, Cray MPI strives to select a NIC device that is closest to the GPU device being used

- Set

```
export MPICH_OFI_NIC_VERBOSE=2
```

  to display information pertaining to NIC selection

- More info in `mpi` man page

Conclusion

BINDING SUMMARY

- Binding is key for performance
- Select the proper CPU binding for MPI tasks and OpenMP threads
  - Use {xthi, hello_jobstep, gpu_check}-like tools to check affinity

- Select the proper GPU affinity
  - E.g. by properly setting `ROCR_VISIBLE_DEVICES` per each MPI task

- Select the proper NIC affinity
  - `export MPICH_OFI_NIC_POLICY=GPU`

Questions?