Advanced Job Launching

mapping applications to hardware

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
  - **CPU**
    The minimum piece of hardware capable of running a PE. It may share some or all of its hardware resources with other CPUs
    Equivalent to a single “Intel Hyperthread”
  - **Compute Unit (CU) or Core**
    The individual unit of hardware for processing. May provide one or more CPUs.

- Software

  Different software approaches also use different naming convention. This is the software-neutral convention we are going to use :

  - **Processing Element (PE)**
    A discrete software process with an individual address space. One PE is equivalent to a UNIX task, MPI Rank, Coarray Image, UPC Thread, or SHMEM PE
  - **Threads**
    A logically separate stream of execution inside a parent PE that shares the same address space (OpenMP, Pthreads)

Launching ESM Parallel applications

- **ALPS : Application Level Placement Scheduler**
- **aprun is the ALPS application launcher**
  - It **must** be used to run applications on the XC compute nodes in ESM mode, (either interactively or as a batch job)
  - If `aprun` is not used, the application will be run on the MOM node (and will most likely fail).
  - `aprun` launches sets of PEs on the compute nodes.
  - `aprun` man page contains several useful examples
  - The 4 most important parameters to set are:

Description	Option
Total Number of PEs used by the application	-n
Number of PEs per compute node	-N
Number of threads per PE (More precise, the “stride” between 2 PEs on a node)	-d
Number of to CPUs to use per Compute Unit	-j

Running applications on the Cray XC30: Some basic examples

Assuming an XC node with 12 core Intel processor

- Each node has: 48 CPUs/Hyperthreads and 24 Compute Units/cores

- **Launching a basic MPI application:**
  - Job has 1024 total ranks/PEs, using 1 CPU per Compute Unit meaning a maximum of 24 PEs per node.

```
$ aprun -n 1024 -N 24 -j1 ./a.out
```

- **To launch the same MPI application but spread over twice as many nodes**

```
$ aprun -n 1024 -N 12 -j1 ./a.out
```

- Can be used to increase the available memory for each PE

- **To use all availble CPUs on a single node**
  - (maximum now 48 PEs per node)

```
$ aprun -n 1024 -N 48 -j2 ./a.out
```

Some examples of hybrid invocation

- **To launch a Hybrid MPI/OpenMP application:**
  - 1024 total ranks, using 1 CPU per Compute Unit (Max 24 Threads)
  - Use 4 PEs per node and 6 Threads per PE
  - Threads set by exporting `OMP_NUM_THREADS`

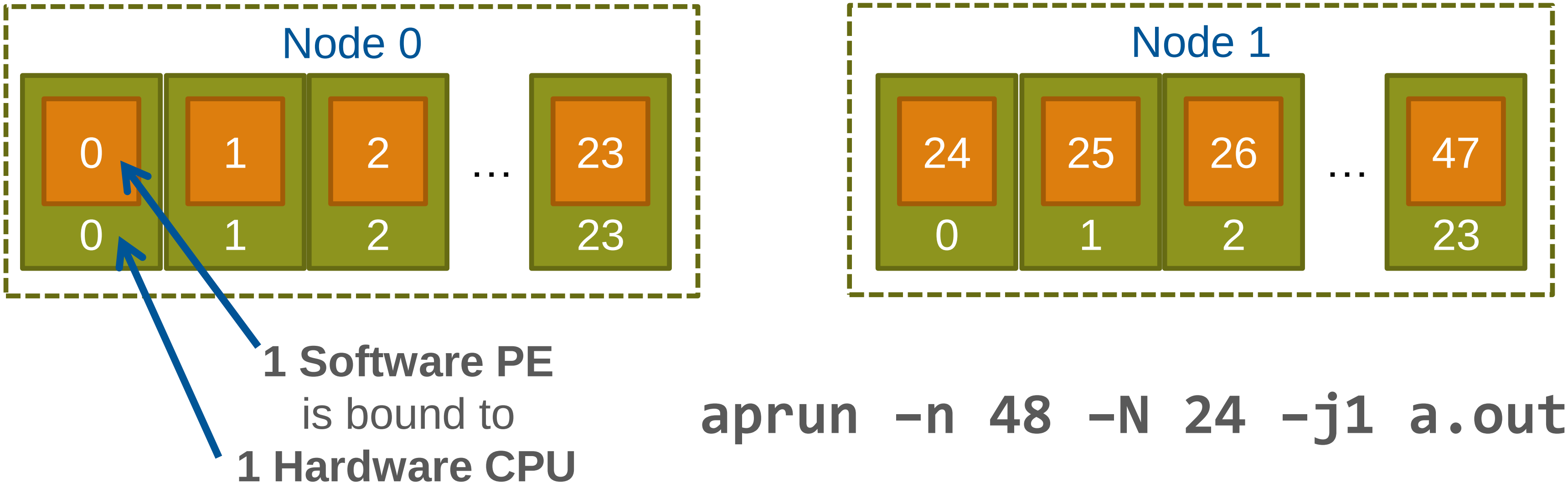
```
$ export OMP_NUM_THREADS=6
$ aprun -n 1024 -N 4 -d $OMP_NUM_THREADS -j1 ./a.out
```

- **Launch the same hybrid application with 2 CPUs per CU**
  - 1024 total ranks, using 2 CPU per Compute Unit (Max 48 Threads)
  - Use 4 PEs per node and 12 Threads per PE

```
$ export OMP_NUM_THREADS=12
$ aprun -n 1024 -N 4 -d $OMP_NUM_THREADS -j2 ./a.out
```

Default Binding - CPU

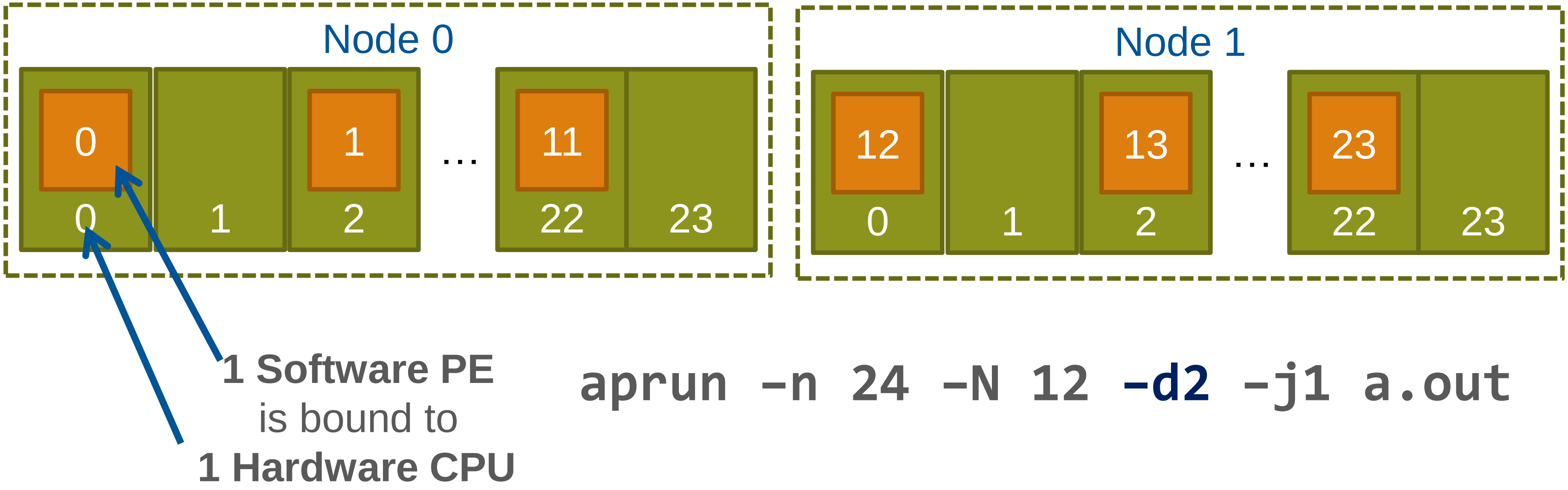
- By default `aprun` will bind each PE to a single CPU for the duration of the run.
- This prevents PEs moving between CPUs.
- All child processes of the PE are bound to the same CPU
- PEs are assigned to CPUs on the node in increasing order from 0. e.g.



```
aprun -n 48 -N 24 -j1 a.out
```

Default Thread Binding (pt 1)

- You can inform aprun how many threads will be created by each PE by passing arguments to the –d (depth) flag.
- aprun does not create threads, just the master PE.
- PEs are bound to CPU spaced by the depth argument, e.g

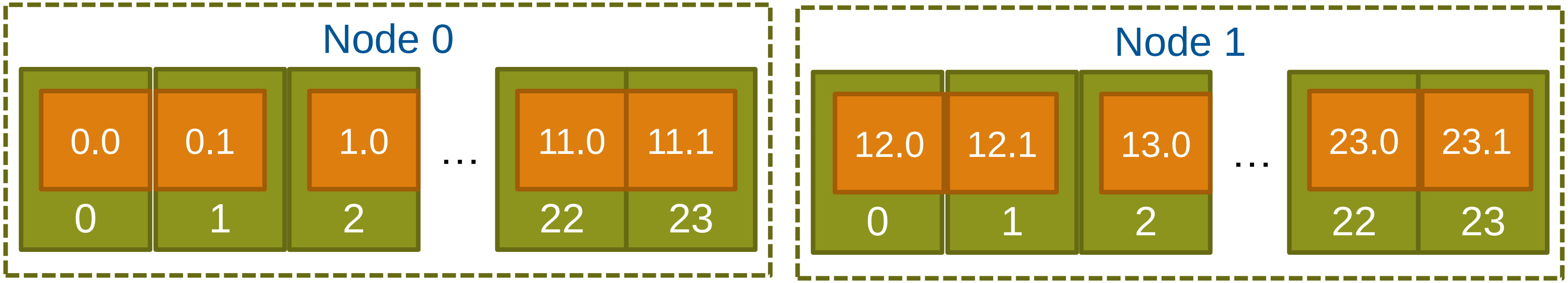


```
aprun –n 24 –N 12 –d2 –j1 a.out
```

Default Thread Binding (pt 2)

- **Each subsequently created child processes/thread is bound by the OS to the next CPU** ***(modulo by the depth argument).*** **e.g.**

```
OMP_NUM_THREADS=2
aprun -n 24 -N 12 -d2 -j1 a.out
```

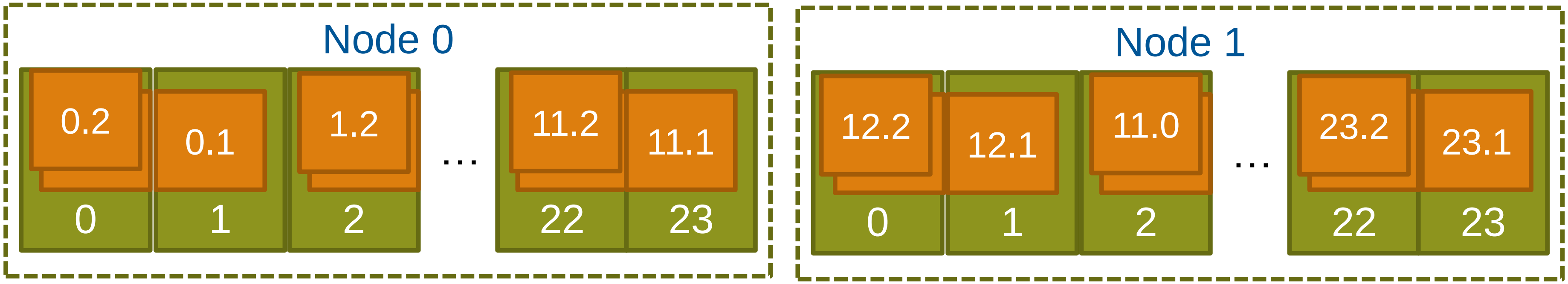


- **Each PE becomes the master thread and spawns a new child thread. The OS binds this child thread to the next CPU.**

Default Thread Binding (pt 3)

- aprun cannot prevent PEs from spawning more threads than requested
- In such cases threads will start to “wrap around” and be assigned to earlier CPUs.
- e.g.

```
OMP_NUM_THREADS=3
aprun -n 24 -N 12 -d2 -j1 a.out
```

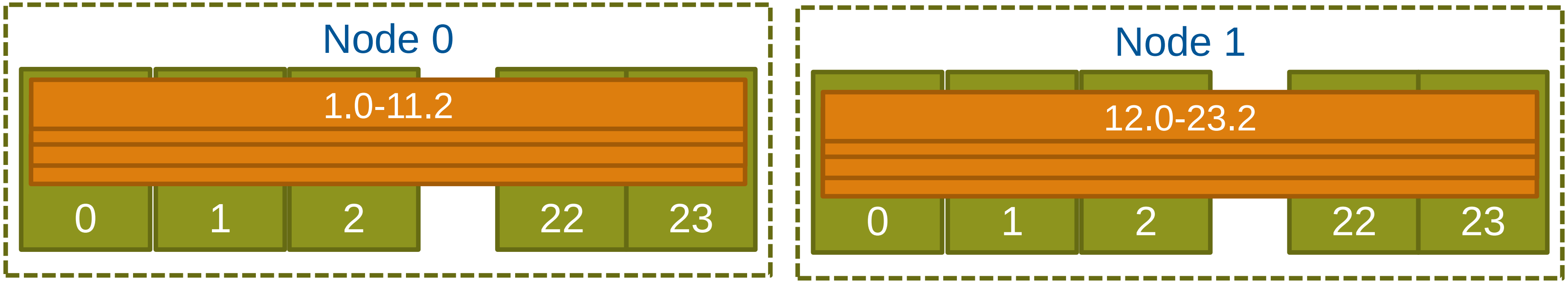


- In this case, the third thread is assigned to the same CPU as the master PE causing contention for resources.

Removing binding entirely

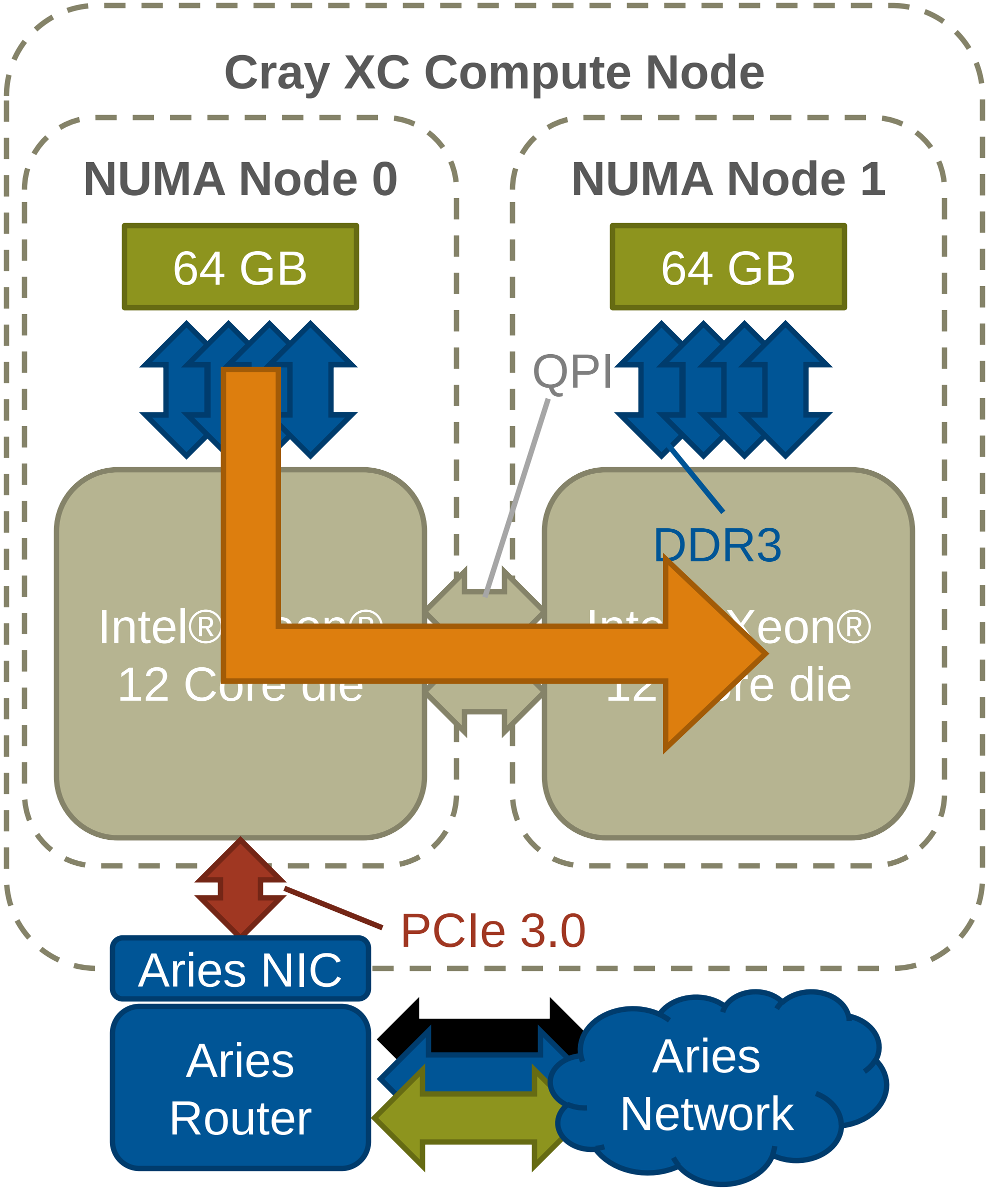
- aprun can be prevented from binding PEs and their children to CPUs, by specifying "–cc none". E.g.

```
        OMP_NUM_THREADS=3
        aprun -n 24 -N 12 -cc none -j1 a.out
```



- All PEs and their child processes and threads are allowed to migrate across cores as determined by the standard Linux process scheduler.
- This is useful where PEs spawn many short lived children (e.g. compilation scripts) or over-subscribe the node.
- (-d removed as it no longer serves a purpose)

NUMA Nodes



The design of the XC node means that CPUs accessing data stored on the other socket/die have to cross the QPI inter-processor bus

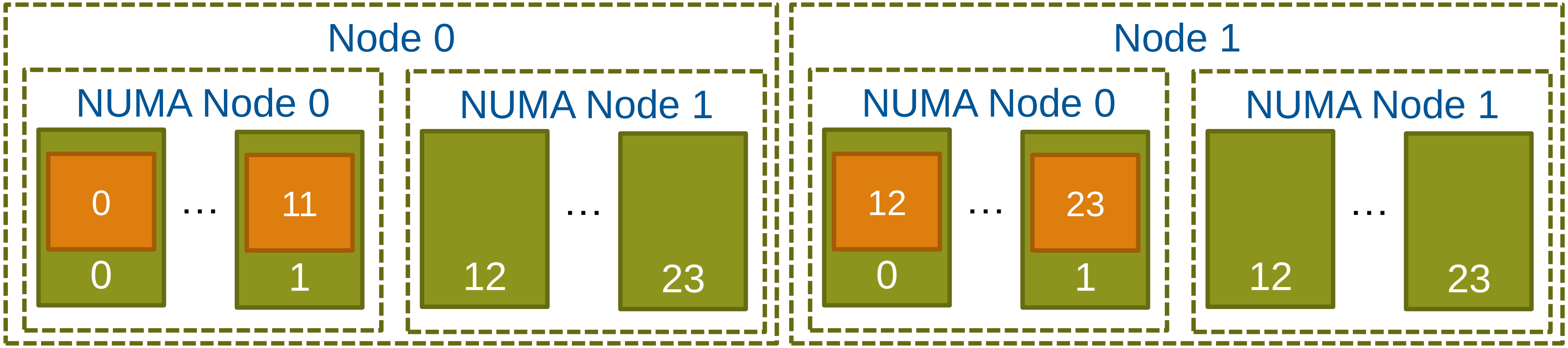
This is marginally slower than accessing local memory and creates “Non-Uniform Memory Access” (NUMA) regions.

Each XC node is divided into two NUMA nodes, associated with the two sockets/dies.

NUMA nodes and CPU binding (pt 1)

- Care has to be taken when under-populating node (running fewer PEs than available CPUs). E.g.

```
aprun -n 24 -N 12 -j1 a.out
```

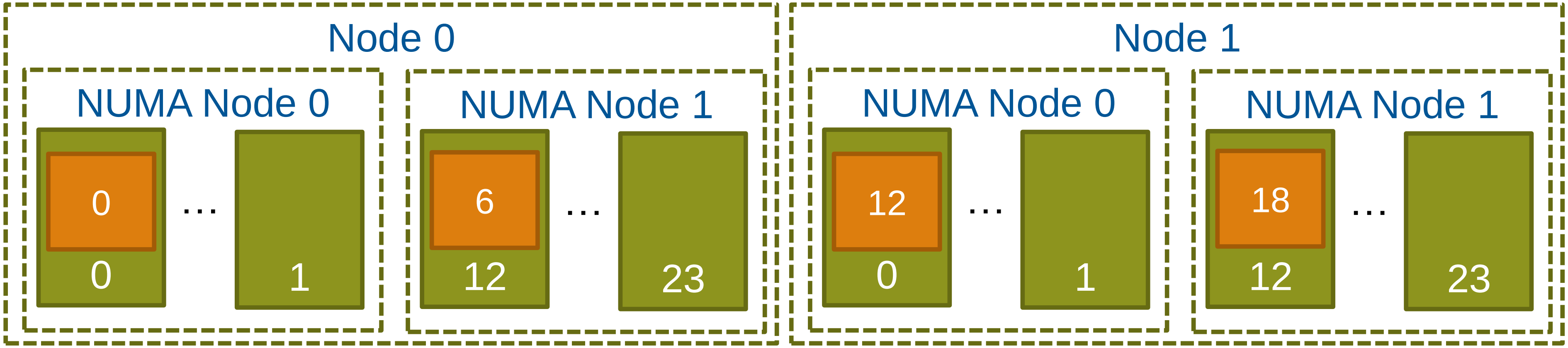


- The default binding will bind all PEs to CPUs in the first NUMA node of each node.
- This will unnecessarily push all memory traffic through only one die’s memory controller. Artificially limiting memory bandwith.

NUMA nodes and CPU binding (pt 2)

- The `-S <PEs>` flag tells `aprun` to distribute that many PEs to each NUMA node, thus evening the load.

```
aprun -n 24 -N 12 -S 6 -j1 a.out
```

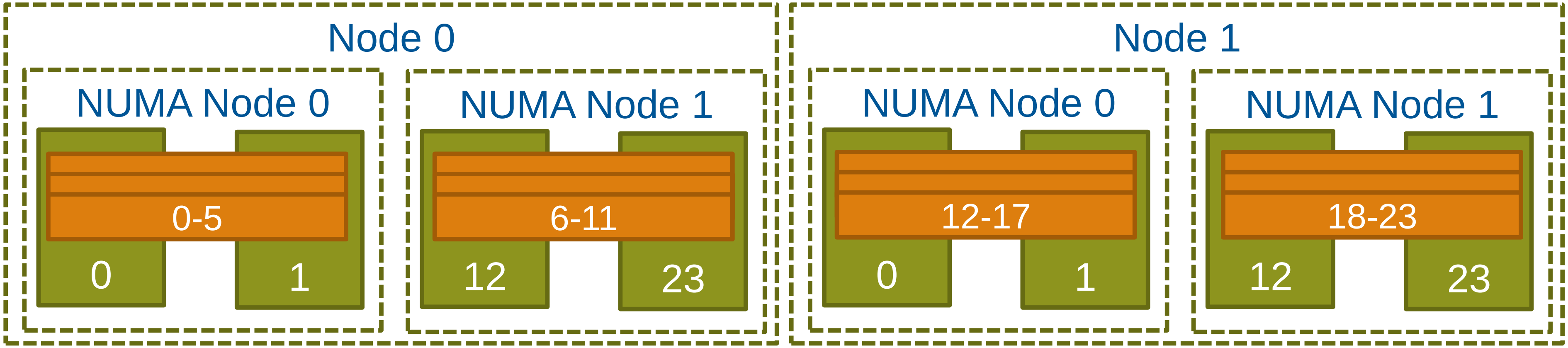


- PEs will be assigned to CPUs in the NUMA node in the standard order, e.g. 0-5 & 12-17. However all CPUs within a NUMA node are essentially identical so there are no additional imbalance problems.

Binding to NUMA nodes

- As well as completely removing binding, it is also possible to make aprun bind PEs to all the CPUs on a NUMA node.

```
aprun -n 24 -N 12 -S 6 -j1 -cc numa_node a.out
```



- PEs will be assigned to the NUMA node that their original PE would have been assigned to with CPU binding and the same options.
- OS allowed to migrate processes within the NUMA node, should be better performance than no binding. "`-cc none`"

Be aware – Intel Helper Threads

- **The Intel OpenMP runtime creates more threads than you might expect**
  - It creates an extra helper thread (OMP_NUM_THREADS+1 threads int total)
  - It also has it's own method of binding to CPUs (KMP_AFFINITY)

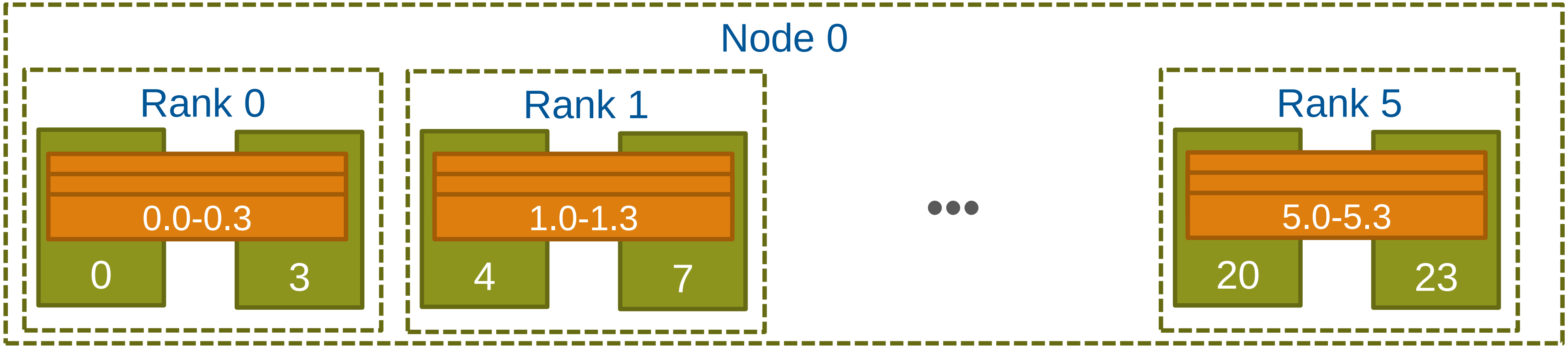
- **Unfortunately both of these options can make things more complicated due to the interactions with CLE binding**

- **Cray advice…**
  - Don't use KMP_AFFINITY to bind threads:
  - `export KMP_AFFINITY=disabled`
  - `use one of the following options:`
    - `aprun -cc [numa_node|none|`**`depth`**`] <exe>`
    - `aprun -cc 0,x,1,…     (the x means don't bind)`

Binding to a CPU set: -depth

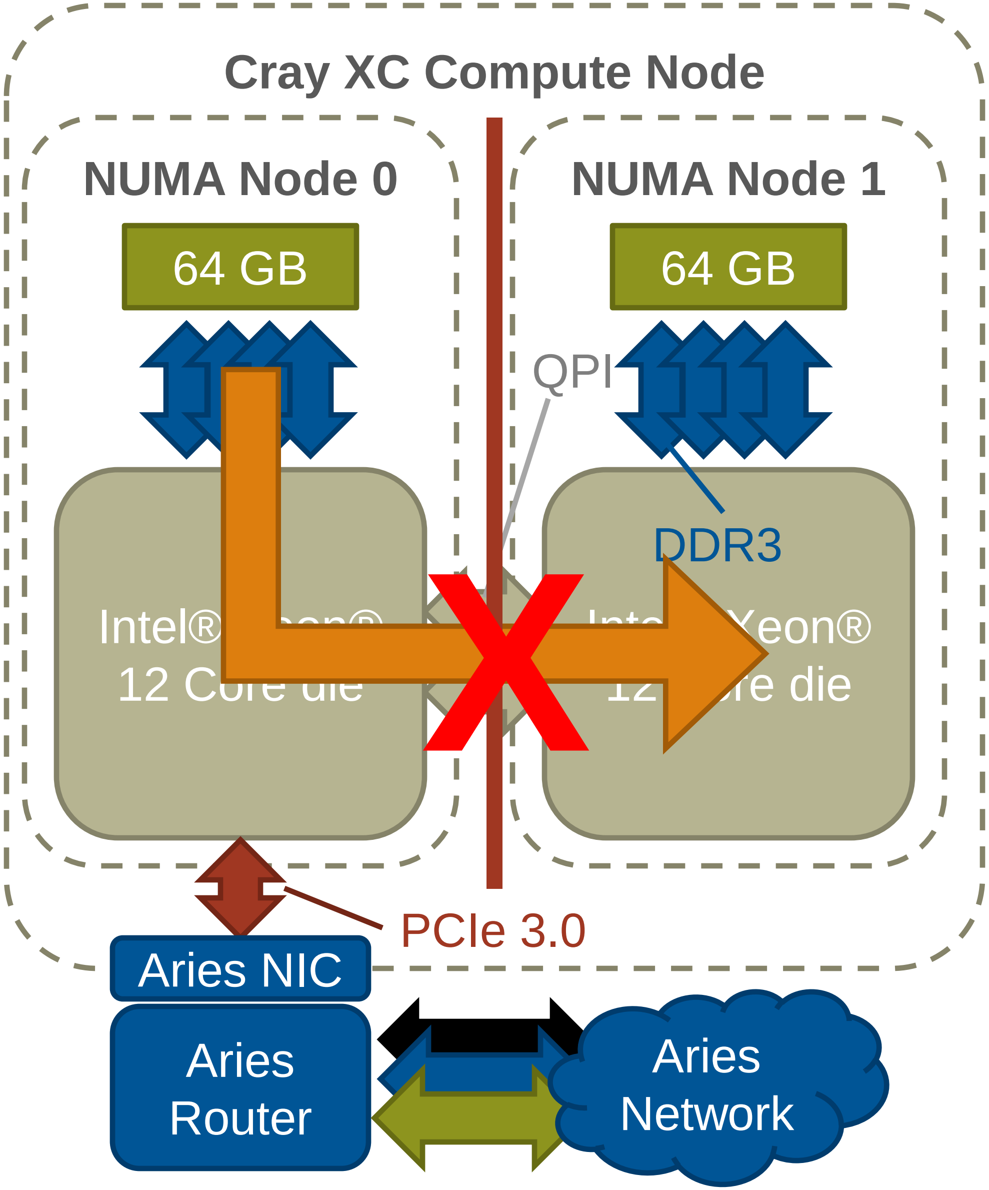
- An extension to “numa_node” is the option -cc depth.
- depth defines that a ‘cpu set’ should be used where all threads belonging to a rank are “unbound”.
  The size of the cpu set is given by the –d option

```
aprun -n 6 -d4 -cc depth -j1 a.out
```



- Solves the ‘Intel Helper Thread’ issue and also the ‘oversubscribing’ of threads.

Strict Memory Containment



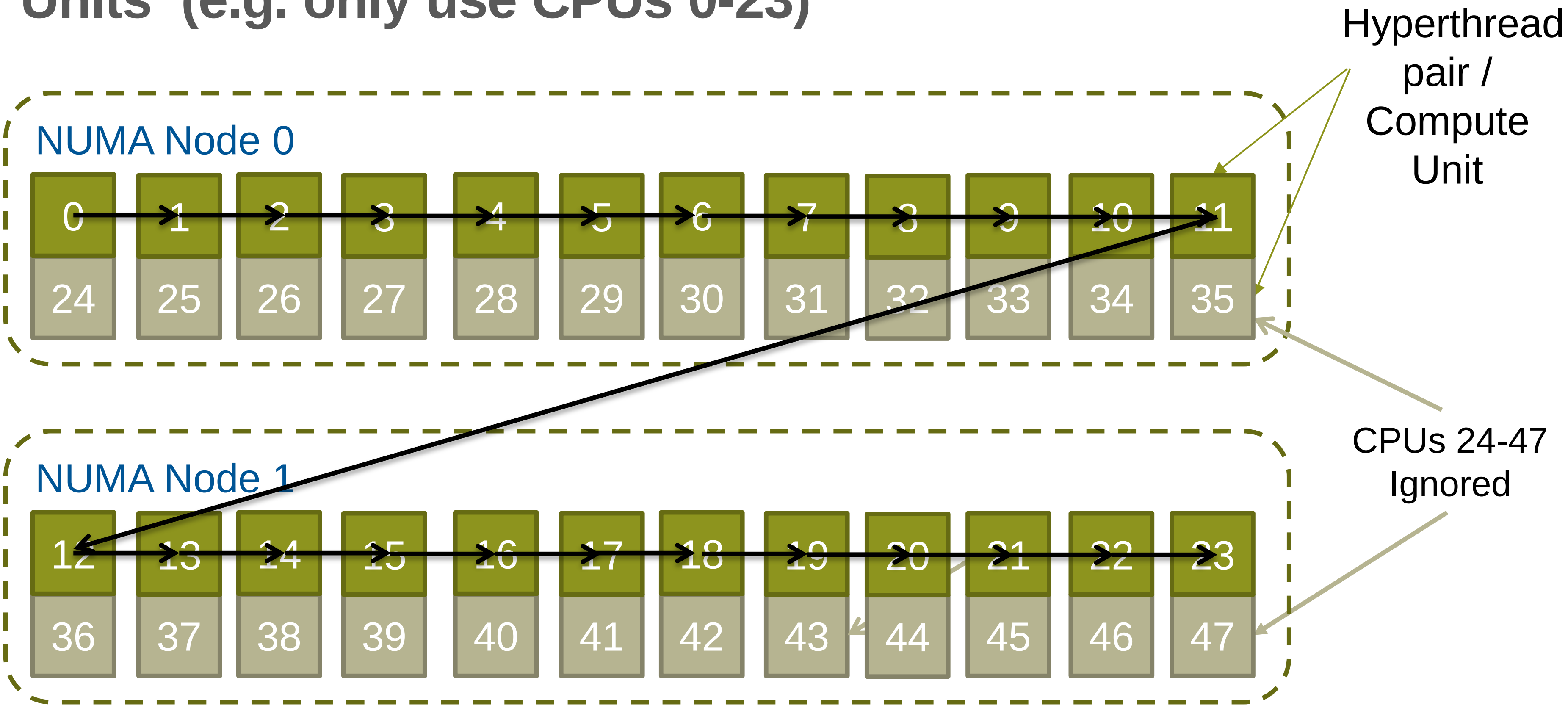
- Each XC node is an shared memory device.
- By default all memory is placed on the NUMA node of the first CPU to “touch” it.
- However, it may be beneficial to setup strict memory containment between NUMA nodes.
- This prevents PEs from one NUMA node allocating memory on another NUMA node.
- This has been shown to improve performance in some applications.

```
aprun -ss -n 48 -N 12\
      -S 6 a.out
```

Ignore Hyperthreads; “-j1” Single Stream Mode

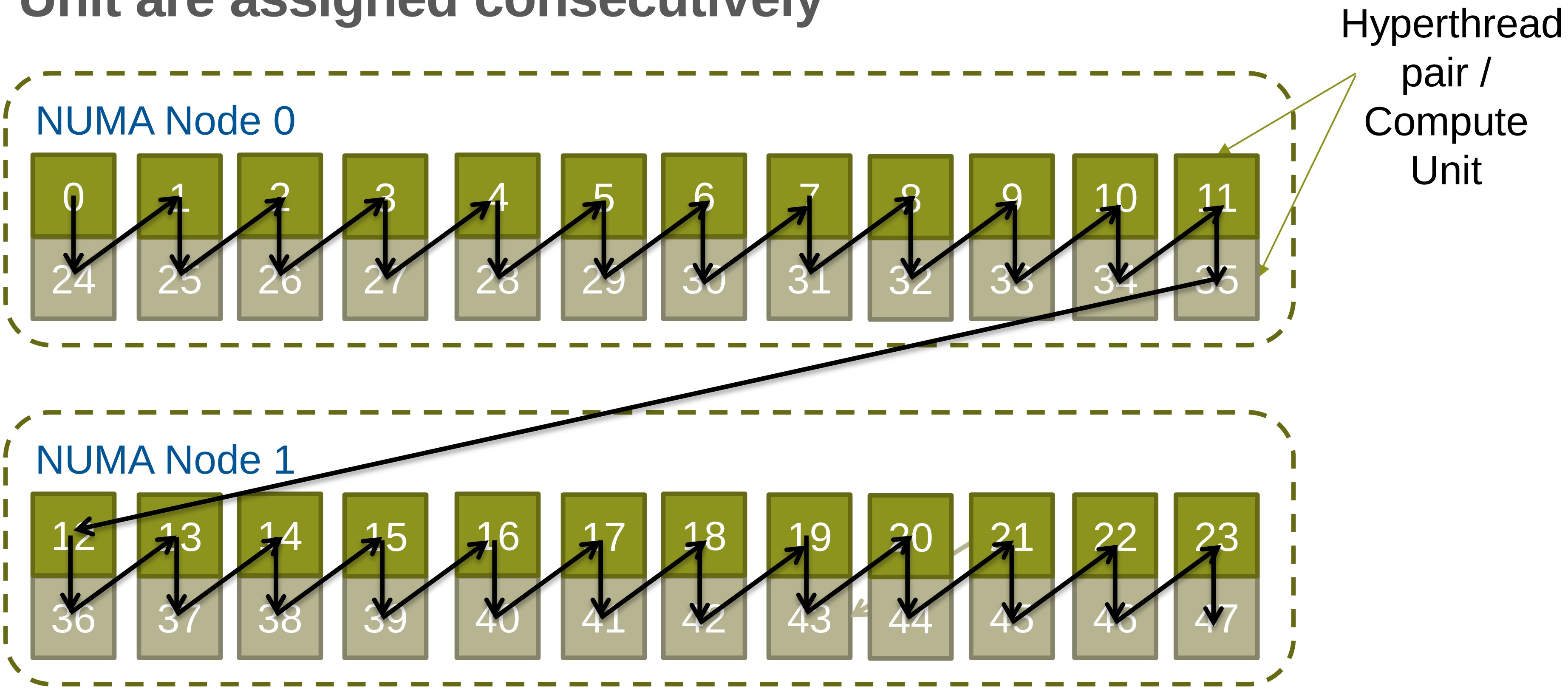
All examples up to now have assumed “`-j1`” or “Single Stream Mode”.

In this mode, `aprun` binds PEs and ranks to the 24 Compute Units (e.g. only use CPUs 0-23)



Include Hyperthreads "-j2" Dual Stream Mode

Specifying "`-j2`" in `aprun` assigns PEs to all of the 48 CPUs available. However CPUs that share a common Compute Unit are assigned consecutively



This means threads will share Compute Units with default binding

Custom Binding

- **aprun also allows users to customise the binding of PEs to CPUs.**
  - Users may pass a colon separated list of CPU binding options to the –cc option.
  - The nth PE on the node is bound by the nth binding option.

- **Each PE binding option may be either a single CPU or a comma separated list of CPUs.**
  - Specifying a single CPU forces the PE and all children and threads to the same PE
  - Specifying a comma separated list binds the PE to the first CPU in the list and children and threads on to the subsequent CPUs (round-robin)
  - Additional PEs will be left unbound.

Custom Binding (example)

- **Custom binding can be hard to get right. The xthi application is useful for testing binding.**
  - Source code available in S-2496 (Workload Management and Application Placement for the Cray Linux Environment) Section 8.7 at docs.cray.com

```
> export OMP_NUM_THREADS=2
> aprun -n 4 -N 16 -cc 3,2:7,8:9,10,4:1 xthi | sort
Hello from rank 0, thread 0, on nid00009. (core affinity = 3)
Hello from rank 0, thread 1, on nid00009. (core affinity = 2)
Hello from rank 1, thread 0, on nid00009. (core affinity = 7)
Hello from rank 1, thread 1, on nid00009. (core affinity = 8)
Hello from rank 2, thread 0, on nid00009. (core affinity = 9)
Hello from rank 2, thread 1, on nid00009. (core affinity = 10)
Hello from rank 3, thread 0, on nid00009. (core affinity = 1)
Hello from rank 3, thread 1, on nid00009. (core affinity = 1)
```

CPU Specialisation

- Despite the low-noise nature of the XC30's CNL Linux OS it occasionally is necessary to run OS/kernel/daemon processes on CPUs.
- If all CPUs are in use then the OS must swap a user process out to execute the OS/kernel/deamon process.
- Normally this introduces only a small amount of noise to the application which evens out over the length of the run.
- However, there are certain pathological cases which amplify these delays if there are frequent synchronisations between nodes (e.g. collectives) preventing scaling.
- CPU specialisation reserves some CPUs for the OS/system/deamon tasks (like OS, MPI progress engines, deamons). This improves overall performance

CPU Specialisation (pt 2)

- On the XC the reserved CPU's are automatically chosen to be from any unused CPUs on Compute Units (e.g. spare Hyperthreads), even if "`-j1`" has been selected.

- You can specify precisely how many free cores/cpus are used using the –r option to reserve them

```
aprun -n 1024 -N 24 -r 8 -j 1 a.out
aprun -n 2048 -N 40 -r 8 -j 2 a.out
```

- Set `MPICH_NEMESIS_ASYNC_PROGRESS` to enabled and note that MPICH_MAX_THREAD_SAFETY should be set to multiple.

  see man mpi

Multiple Program Multiple Data Mode

As well as launching Single Program Multiple Data (SPMD) programs `aprun` is capable of launching programs as Multiple Program Multiple Data (MPMD).

By passing multiple sets of arguments separated by a **colon**, multiple programs can be launched within the same communication framework (e.g. `MPI_COMM_WORLD`). e.g.

```
aprun -n 480 -N 24 atmosphere.exe : -n 96 -N 12 -d 2 ocean.exe
```

Each different group of PEs is launched on their own unique set of nodes. Global ID (e.g. ranks) are assigned in increasing order from left to right (i.e. rank 0 is always part of the first binary).

Some other useful `aprun` options

Option	Description
`-b`	Disable binary transfer. Prevents ALPS from reading the executable on the login node and distributing it to participating compute nodes. Instead each node will read the binary from the filesystem (assuming it is mounted)
`-q`	Quiet mode, suppress all non-fatal error messages.
`-T`	Synchronizes the applications stdout and stderr to prevent interleaving of their output.