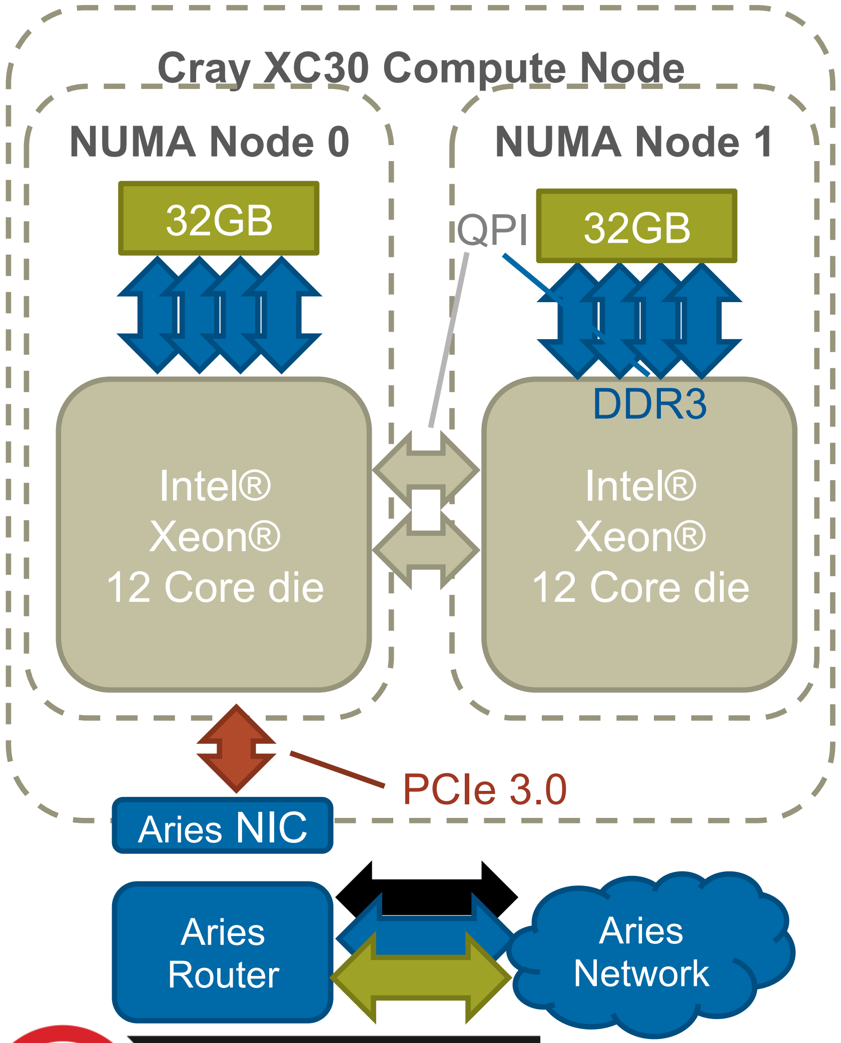
ARCHER Processors

Slides contributed from Cray and EPCC

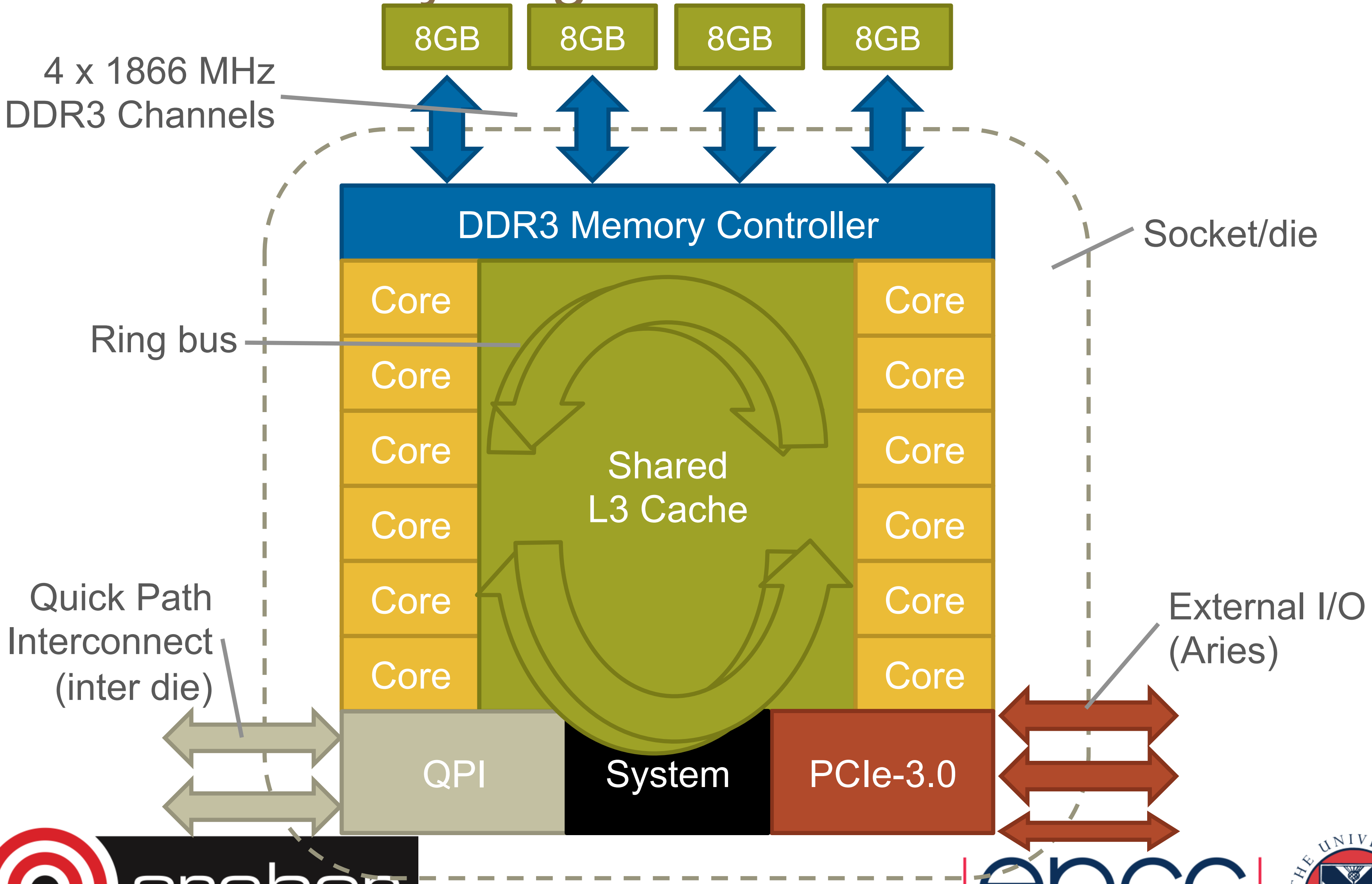
Cray XC30 Intel® Xeon® Compute Node



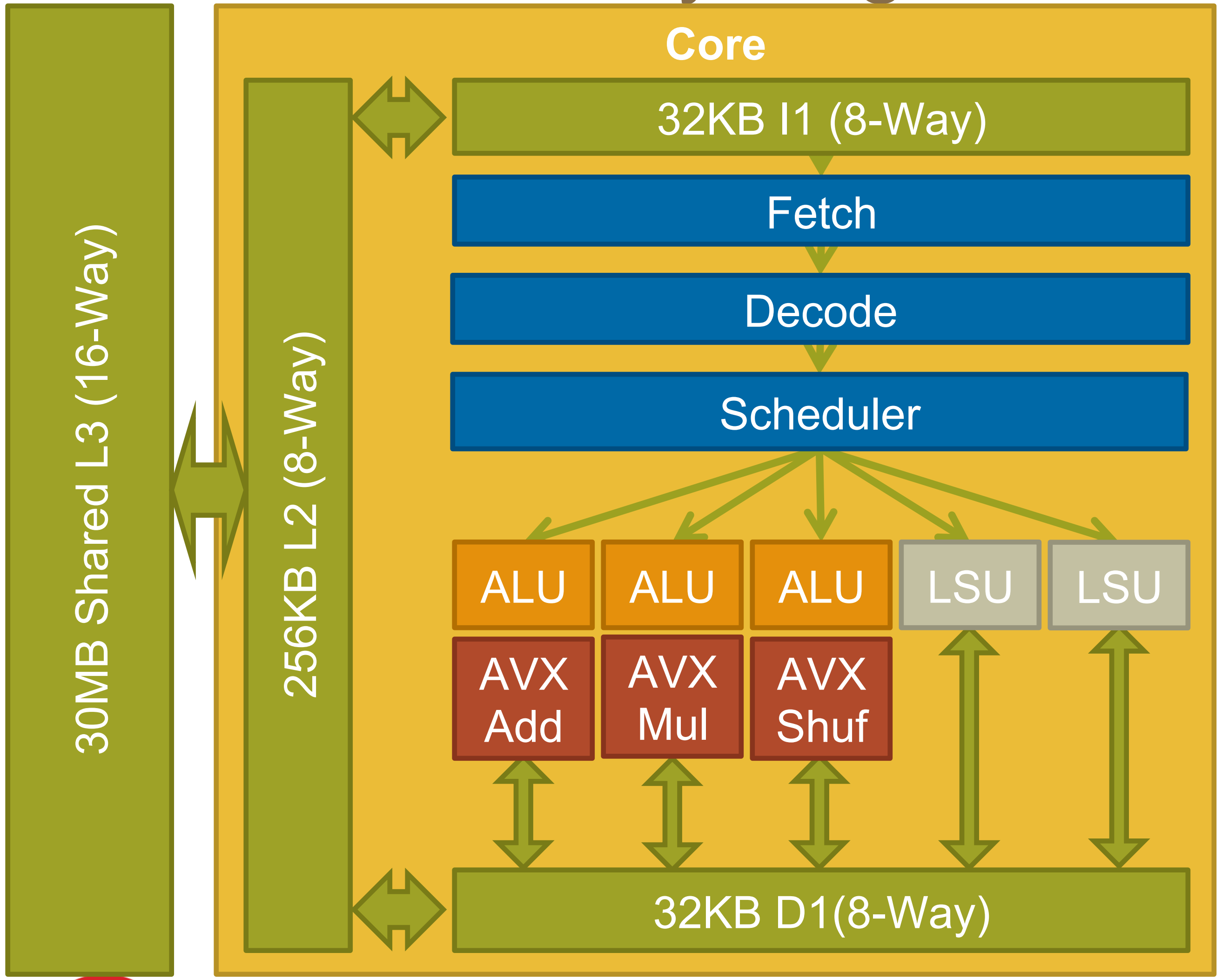
The XC30 Compute node features:

- 2 x Intel® Xeon® Sockets/die
  - 12 core Ivy Bridge
  - QPI interconnect
  - 2.7 GHz (3.5 GHz)
  - Forms 2 NUMA nodes
- 8 x 1833MHz DDR3
  - 8 GB per Channel
  - 64/128 GB total
- 1 x Aries NIC
  - Connects to shared Aries router and wider network
  - PCI-e 3.0

Intel® Xeon® Ivybridge 12-core socket/die



Intel® Xeon® Ivybridge Core Structure

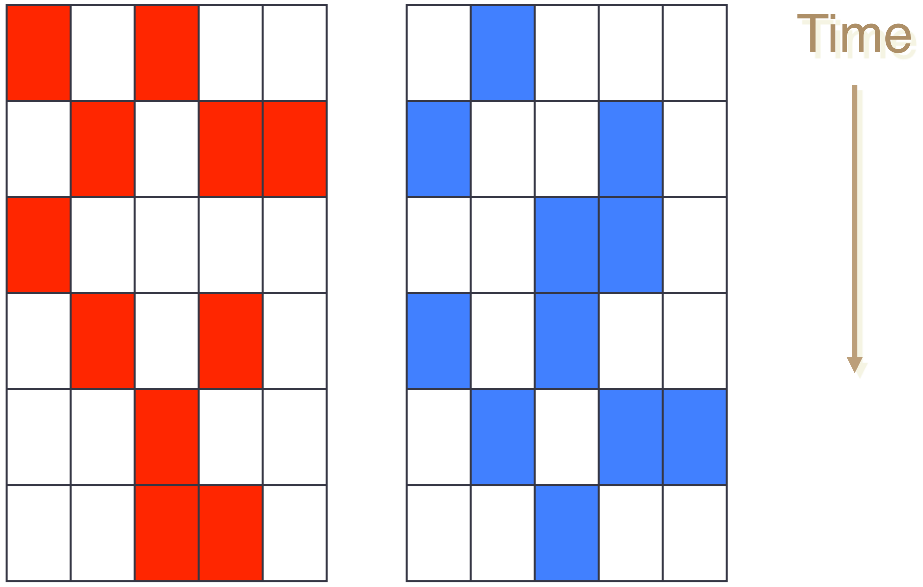


- 256 bit AVX Instructions (4 double precision floating point)
  - 1 x Add
  - 1 x Multiply
  - 1 x Other
- 2 Hardware threads (Hyperthreads)
- Peak DP FP per node 8FLOPS/ clock

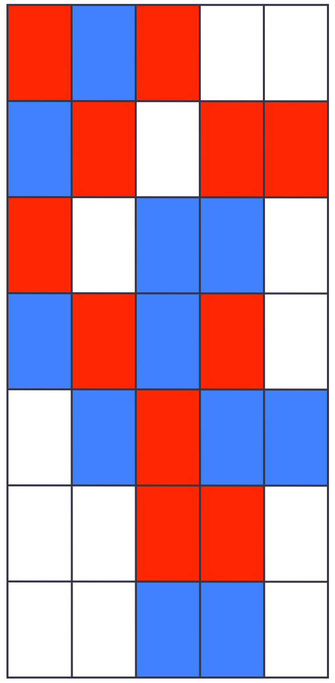
Hyper-threading

- Hyper-threading (or Simultaneous multithreading (SMT)) tries to fill these spare slots by mixing instructions from more than one thread in the same clock cycle.
- Requires some replication of hardware
  - instruction pointer, instruction TLB, register rename logic, etc.
  - Intel Xeon only requires about 5% extra chip area to support SMT
- ...but everything else is shared between threads
  - functional units, register file, memory system (including caches)
  - sharing of caches means there is no coherency problem
- For most architectures, two or four threads is all that makes sense

Hyper-threading example



Two threads on two cores



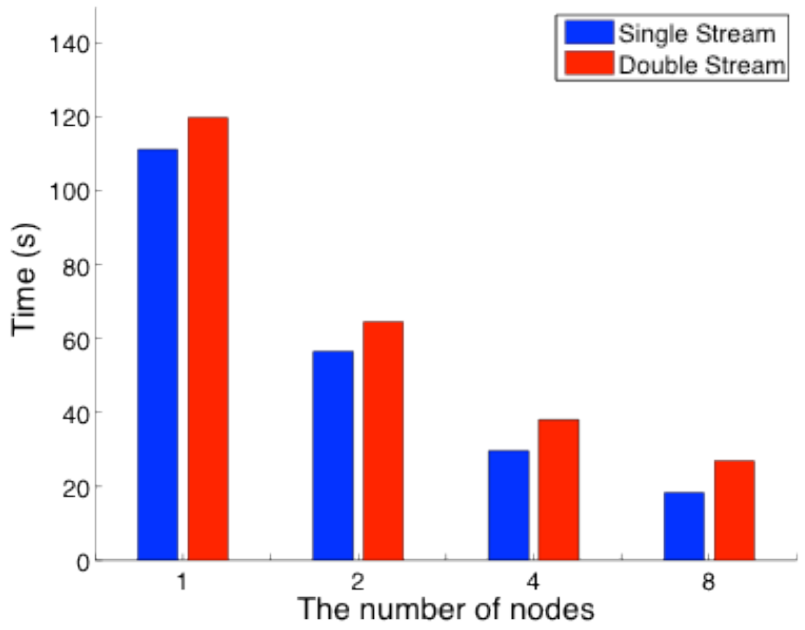
Two threads on one SMT core

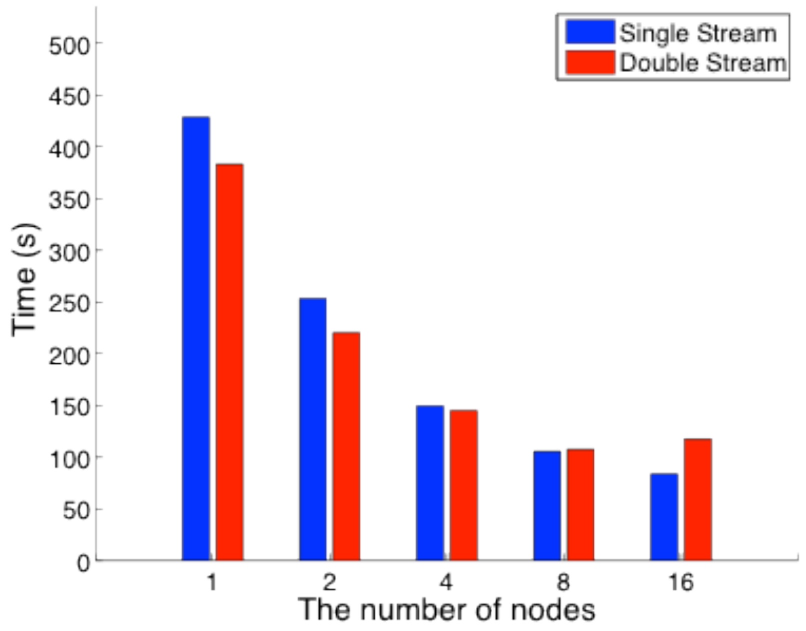
More on Hyper-threading

- How successful is hyper-threading?
  - depends on the application, and how the 2 threads contend for the shared resources.
- In practice, gains seem to be limited to around 1.2 to 1.3 times speedup over a single thread.
  - benefits will be limited if both threads are using the same functional units (e.g. FPUs) intensively.
- For memory intensive code, hyper-threading can cause slow down
  - caches are not thread-aware
  - when two threads share the same caches, each will cause evictions of data belonging to the other thread.

Hyper-threading example performance

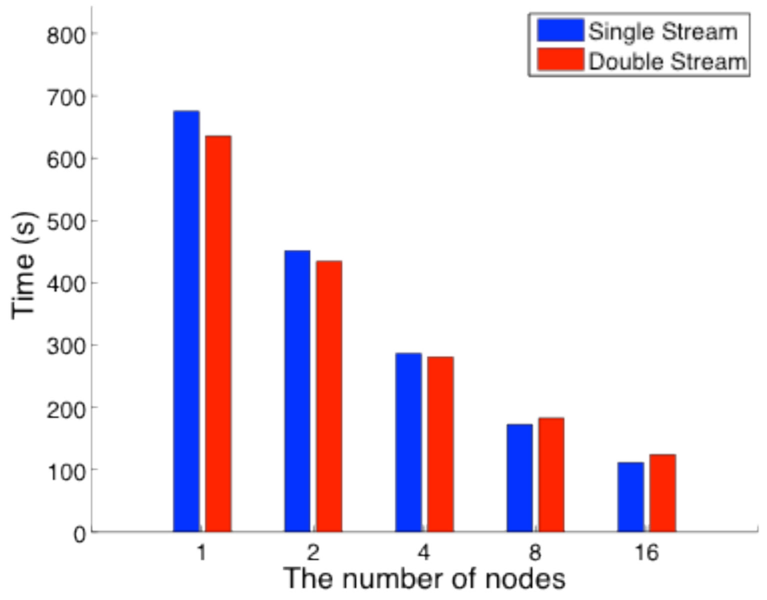
- XC30
  - Sandy-bridge (8 cores)
- VASP
- NAMD



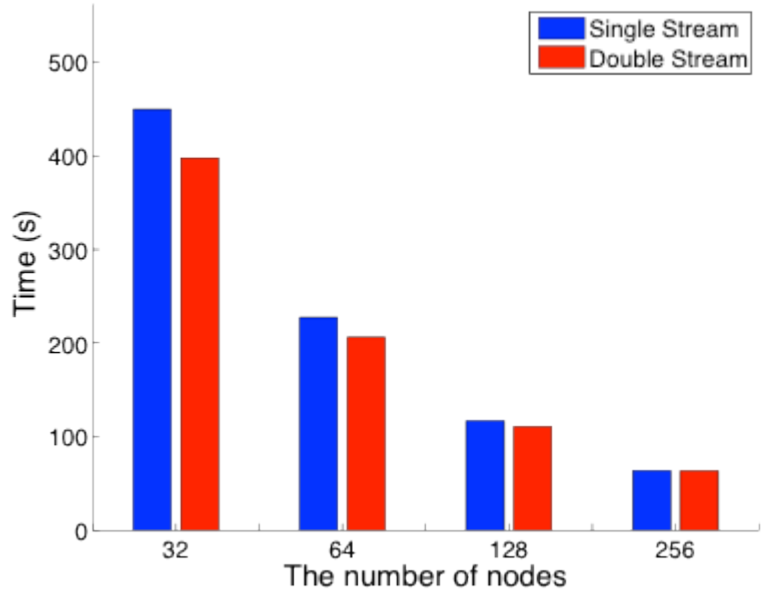


Effects of Hyper-Threading on the NERSC workload on Edison http://www.nersc.gov/assets/CUG13HTpaper.pdf

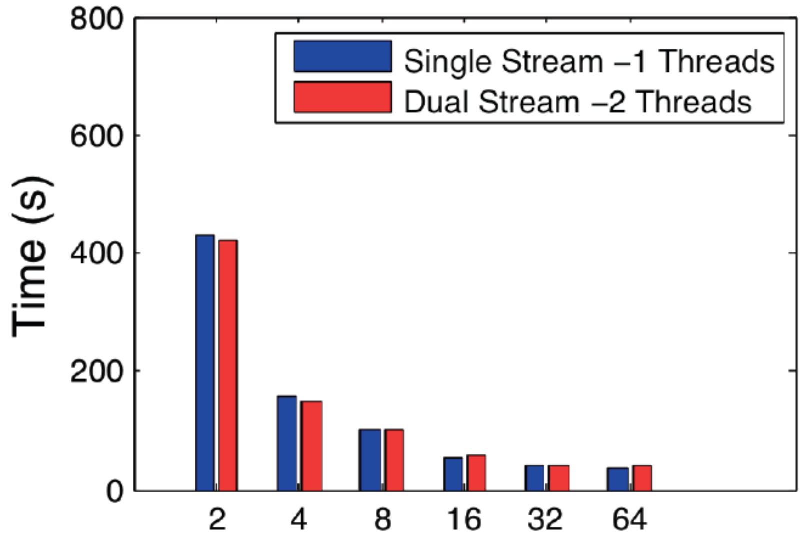
- NWChem



- GTC



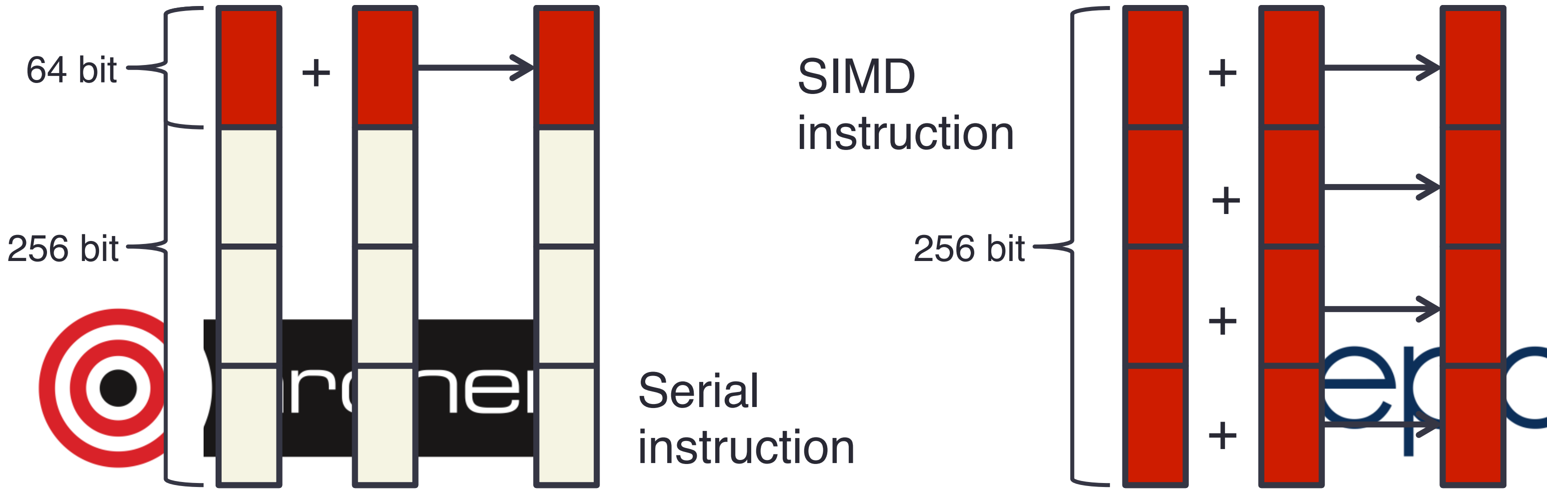
- Quantum Espresso



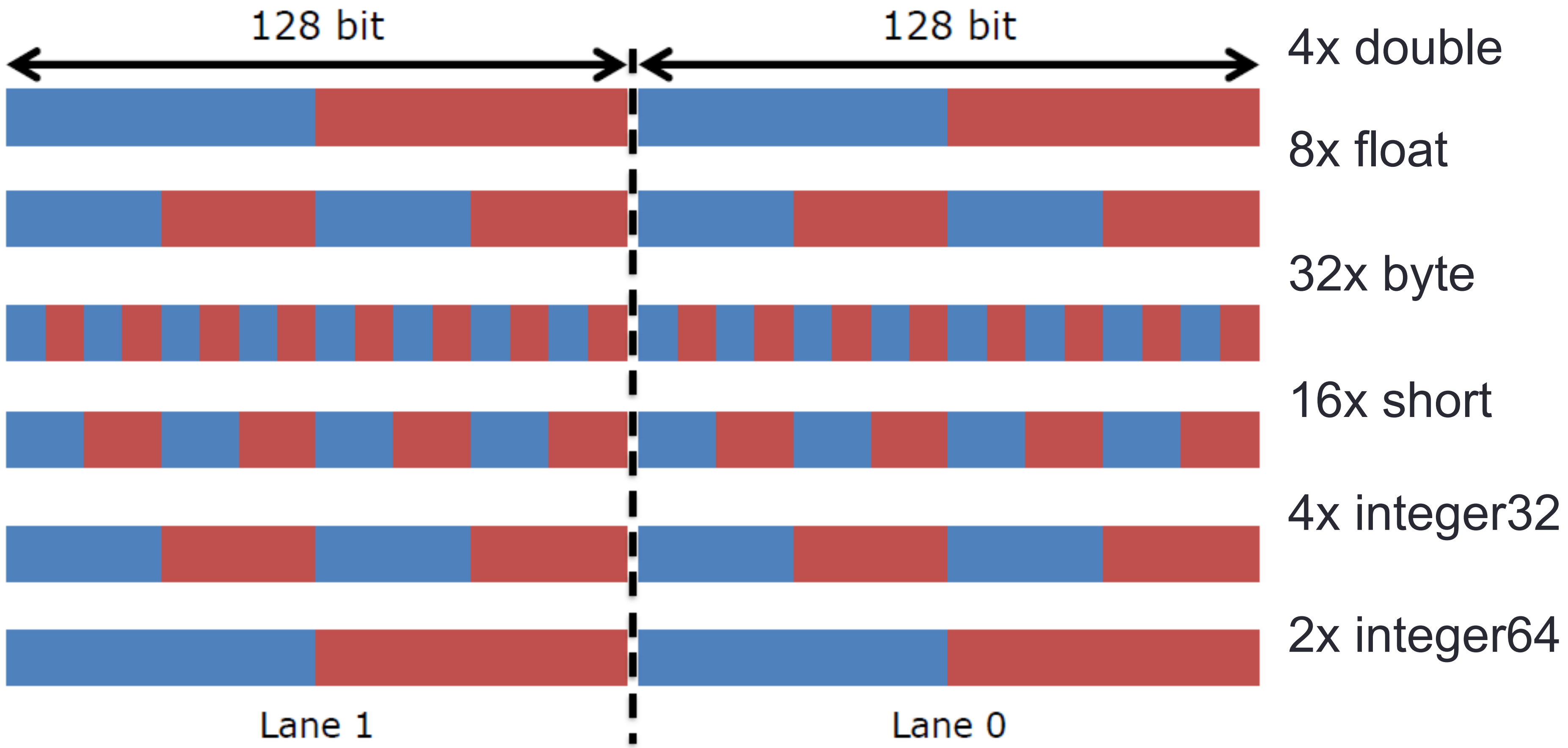
SIMD Vector Operations

- Same operation on multiple data items
  - Wide registers
  - SIMD needed to approach FLOP peak performance, but your code must be capable of vectorisation
- **x86 SIMD instruction sets:**
  - SSE: register width = 128 Bit
    - 2 double precision floating point operands
  - AVX: register width = 256 Bit
    - 4 double precision floating point operands

```
for(i=0;i<N;i++){
    a[i] = b[i] + c[i]
}
do i=1,N
    a(i) = b(i) + c(i)
end do
```



Intel AVX



- **+, – , * gives 2x w.r.t. SSE; / and sqrt same performance**

When does the compiler vectorize

- What can be vectorized
    - Only loops

- Usually only one loop is vectorizable in loopnest
    - And most compilers only consider inner loop

- Optimising compilers will use vector instructions
    - Relies on code being vectorizable
    - Or in a form that the compiler can convert to be vectorizable
        - Some compilers are better at this than others

- Check the compiler output listing and/or assembler listing
    - Look for packed AVX instructions

Helping vectorization

- Is there a good reason for non-vectorization?
  - There is an overhead in setting up vectorization; maybe it's not worth it
    - Could you unroll inner (or outer) loop to provide more work?
- Does the loop have dependencies?
  - information carried between iterations
    - e.g. counter: **`total = total + a(i)`**
  - No:
    - Tell the compiler that it is safe to vectorize
      - !dir$ IVDEP or #pragma ivdep directive above loop (CCE, but works with most compilers)
      - C99: restrict keyword (or compile with `-hrestrict=a` with CCE)
  - Yes:
    - Rewrite code to use algorithm without dependencies, e.g.
      - promote loop scalars to vectors (single dimension array)
      - use calculated values (based on loop index) rather than iterated counters, e.g.
        - Replace: **`count = count + 2; a(count) = ...`**
        - By: **`a(2*i) = ...`**
      - move **`if`** statements outside the inner loop
        - may need temporary vectors to do this (otherwise use **masking operations**)
    - If you need to do too much extra work to vectorize, may not be worth it.

Let's consider a non-vectorizable loop

```
16.   + 1-------<    do j = 1,N
17.     1              x = xinit
18.   + 1 r4----<      do i = 1,N
19.     1 r4             x = x + vexpr(i,j)
20.     1 r4             y(i) = y(i) + x
21.     1 r4---->      end do
22.     1------->    end do
```

Look further down for associated messages

1.497ms

ftn-6254 ftn: VECTOR File = bufpack.F90, Line = 16

A loop starting at line 16 was **not vectorized** because a recurrence was found on "y" at line 20.

ftn-6005 ftn: SCALAR File = bufpack.F90, Line = 18

A loop starting at line 18 was **unrolled 4 times**.

ftn-6254 ftn: VECTOR File = bufpack.F90, Line = 18

A loop starting at line 18 was not vectorized because a recurrence was found on "x" at line 19.

For more info, type:
`explain ftn-6254`

Now make a small modification

```
38.     Vf------<    do i = 1,N
39.     Vf             x(i) = xinit
40.     Vf------>    end do
41.
42.     ir4-----<    do j = 1,N
43.     ir4 if--<      do i = 1,N
44.     ir4 if           x(i) = x(i) + vexpr(i,j)
45.     ir4 if           y(i) = y(i) + x(i)
46.     ir4 if-->      end do
47.     ir4----->    end do
```

```
ftn-6007 ftn: SCALAR File = bufpack.F90, Line = 42
  A loop starting at line 42 was interchanged with the loop starting at line 43.
ftn-6004 ftn: SCALAR File = bufpack.F90, Line = 43
  A loop starting at line 43 was fused with the loop starting at line 38.
ftn-6204 ftn: VECTOR File = bufpack.F90, Line = 38
  A loop starting at line 38 was vectorized.
ftn-6208 ftn: VECTOR File = bufpack.F90, Line = 42
  A loop starting at line 42 was vectorized as part of the loop starting at line 38.
ftn-6005 ftn: SCALAR File = bufpack.F90, Line = 42
  A loop starting at line 42 was unrolled 4 times.
```

x promoted to vector:

- trade slightly more memory
- for better performance

1.089ms

-37%

N.B. outer loop vectorization here

When does the Cray Compiler vectorize?

- The Cray compiler will only vectorize loops
  - Constant strides are best, indirect addressing is bad
    - Scatter/gather operations (not implemented in AVX)
  - Can vectorize across inlined functions
  - Needs to know loop tripcount (but only at runtime)
    - do/while loops should be avoided
  - No recursion allowed
    - if you have this, consider rewriting the loop
  - If you can't vectorize the entire loop, consider splitting it
    - so as much of the loop is vectorized as possible

- Always check the compiler output to see what it did
  - CCE: `-hlist=a`
  - Intel: `-vec-report[0..5]`
  - GNU: `-ftree-vectorizer-verbose=1`
  - or (for the hard core) check the assembler generated

- Clues from CrayPAT's HWPC measurements
  - `export PAT_RT_HWPC=13` or `14` # Floating point operations SP,DP
  - Complicated, but look for ratio of operations/instructions > 1
    - expect 4 for pure AVX with double precision floats

Intel TurboBoost

- Operating frequency of Processor can change
  - 2.7 GHz base frequency
  - 3.5 GHz maximum frequency
  - Increments of 0.1 GHz
- E5-2697v2
  - Turbo modes: 3/3/3/3/3/3/3/4/5/6/7/8
  - 6-12 cores active, maximum frequency 3.0 GHz
  - 0.1 GHz increase for each core not active above this
- System automatically changes, based on:
  - Number of active cores
  - Estimated current consumption
  - Estimated power consumption
  - Processor temperature

Glossary of Cray terminology

PE/Processing Element

- A discrete software process with an individual address space. One PE is equivalent to1 MPI Rank, 1 Coarray Image, 1 UPC Thread, or 1 SHMEM PE

Threads

- A logically separate stream of execution inside a parent PE that shares the same address space

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

- The individual unit of hardware for processing, may be seen described as a “core”.

Running applications on the Cray XC30: Some basic examples

Assuming an XC30 node with 12 core Ivybridge processors

- 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.

```
#PBS -l select=43
$ aprun –n 1024 –N 24 –j1 ./a.out
```

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

```
#PBS -l select=86
$ aprun –n 1024 –N 12 –j1 ./a.out
```

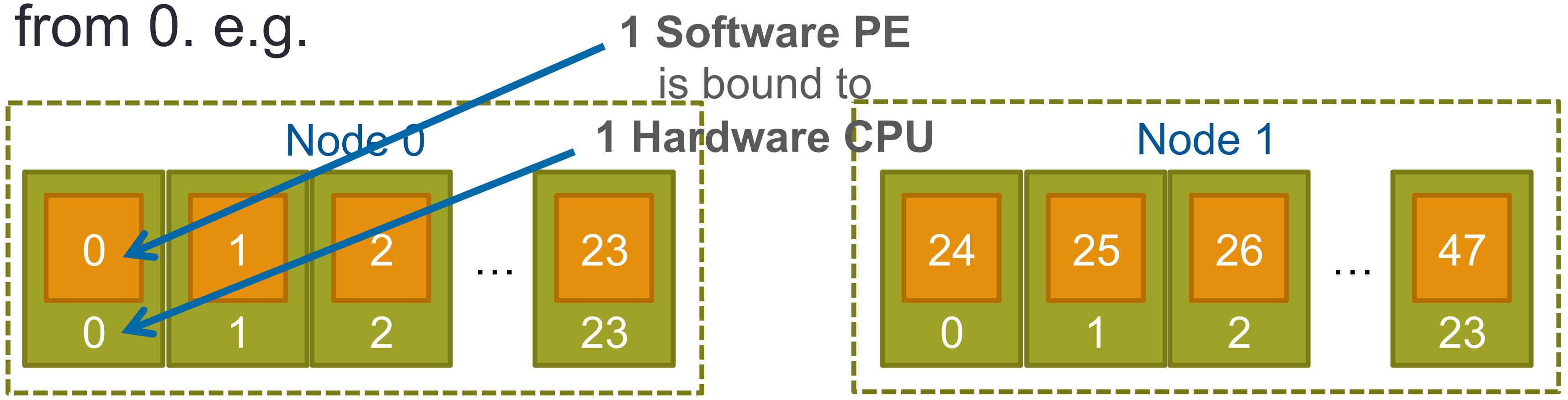
  - Can be used to increase the available memory for each PE

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

```
#PBS -l select=22
$ aprun –n 1024 –N 48 –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.

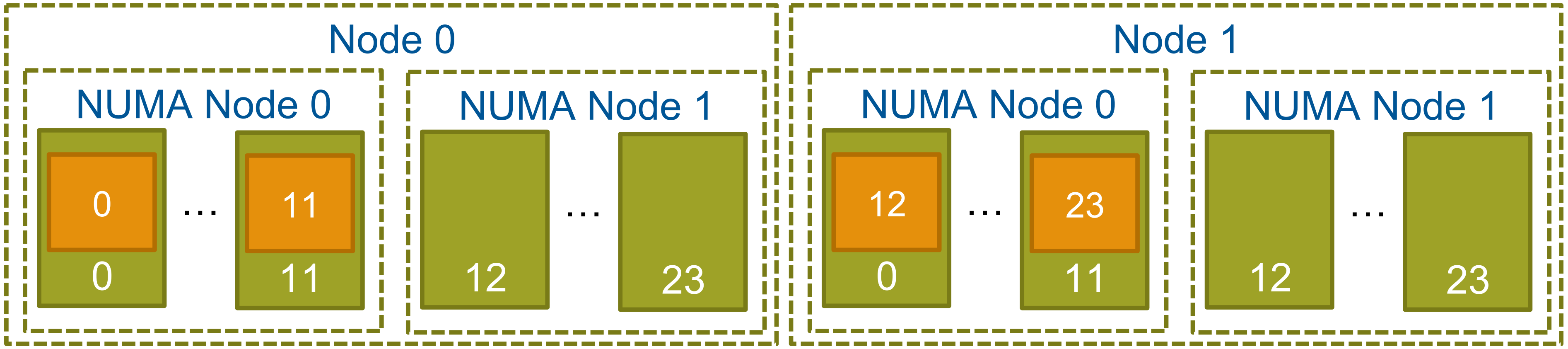


```
aprun -n 48 -N 24 -j1 a.out
```

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
```

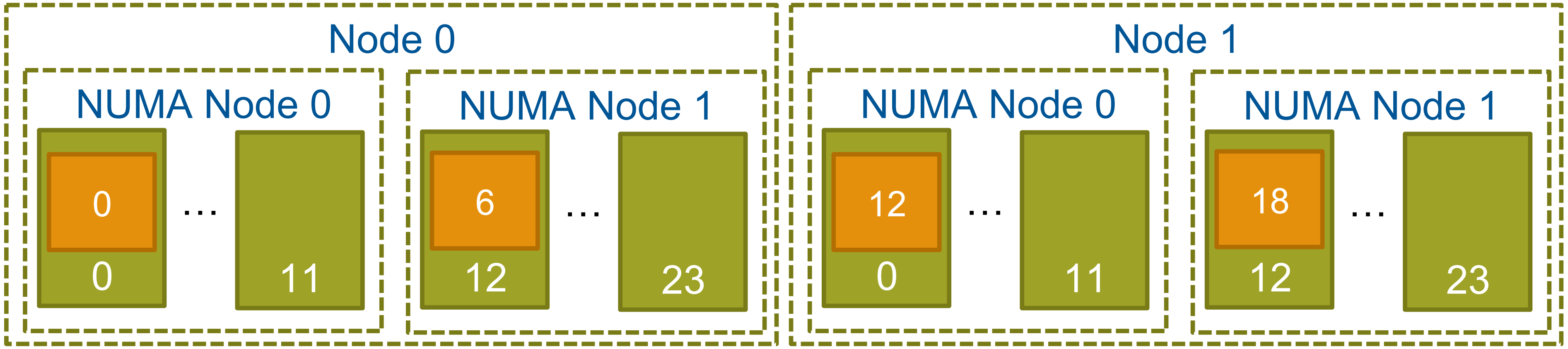


- The default binding will bind all PEs to CPUs in the first NUMA node of each node.
- This will unnecessarily push all memory traffic through only one die's memory controller. Artificially limiting memory bandwidth.

NUMA nodes and CPU binding (pt 2)

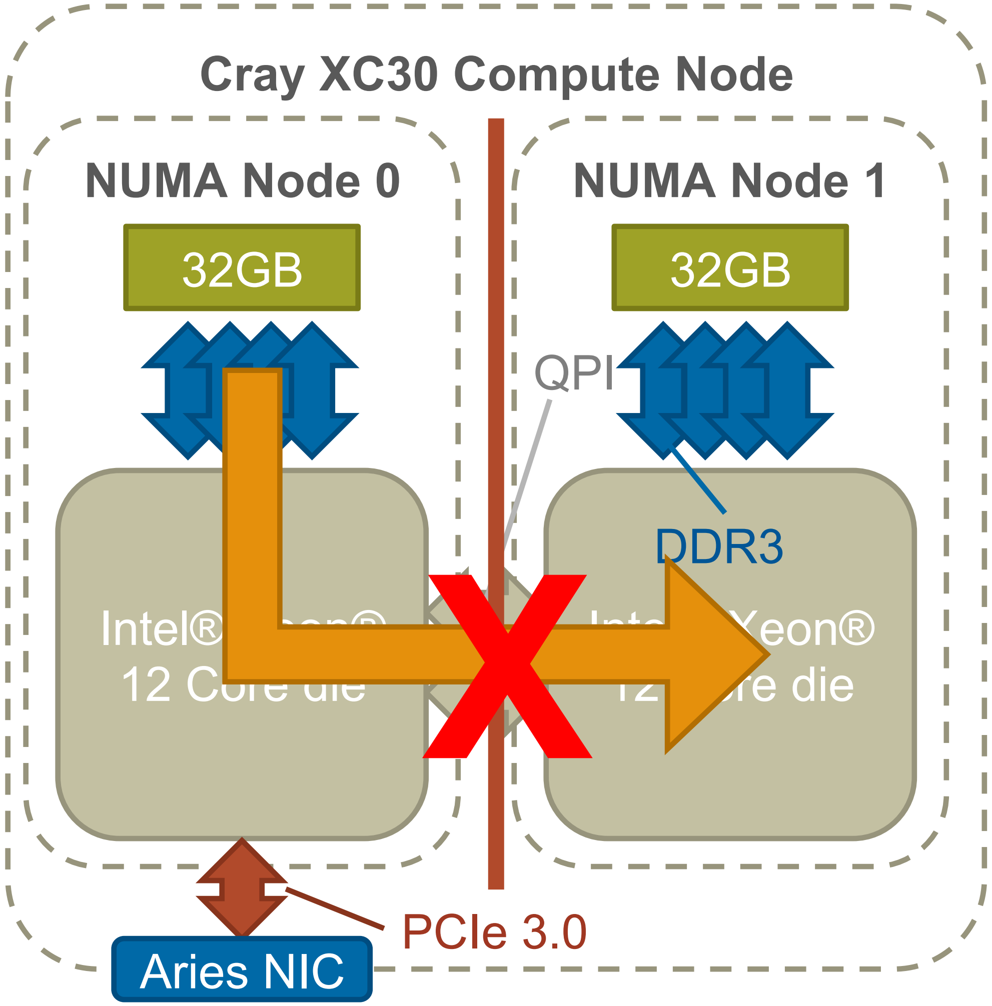
- The `-S <PEs>` flag tells `aprun` to distribute that many PEs to each NUMA node, thus evening the load.

```
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.

Strict Memory Containment



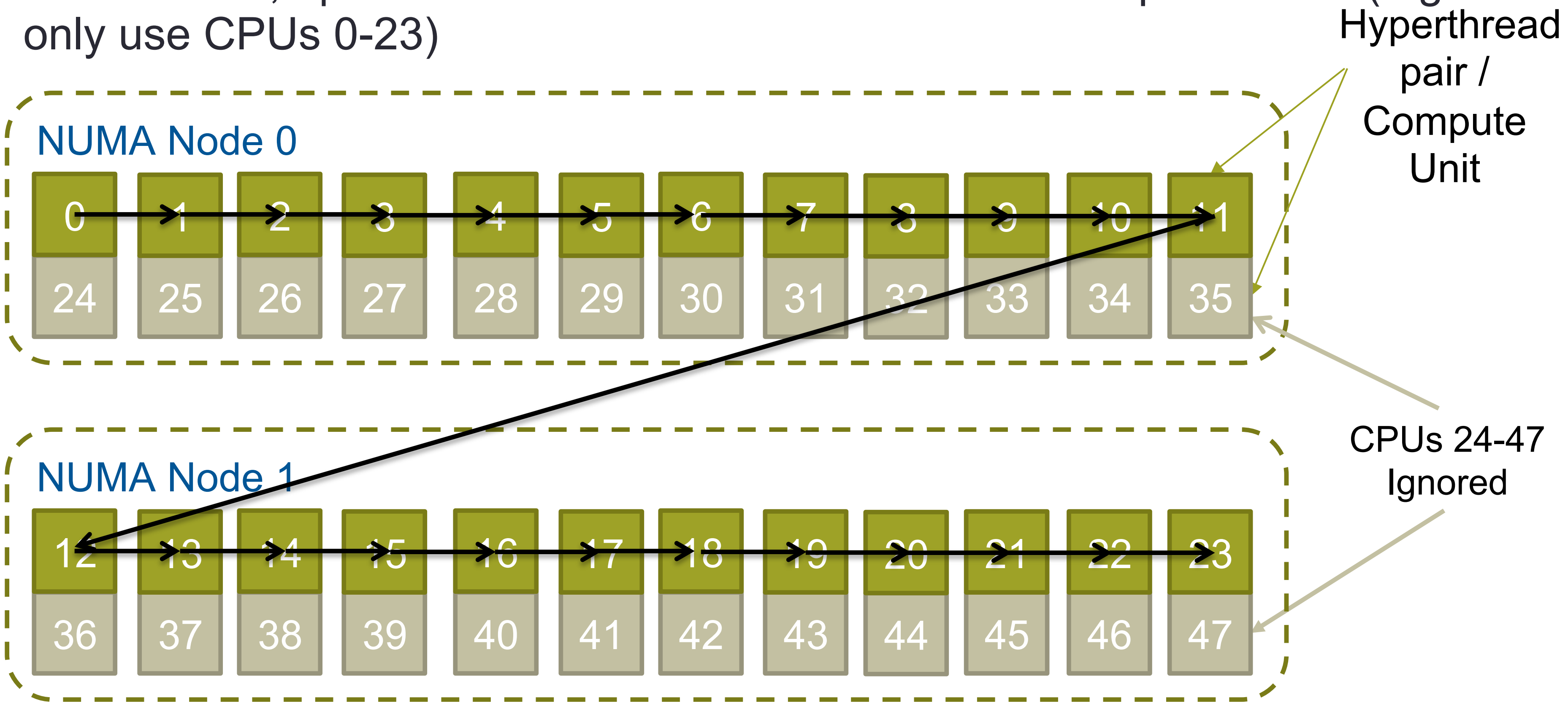
- Each XC30 node is an shared memory device.
- By default all memory is placed on the NUMA node of the first CPU to “touch” it.
- However, it may be beneficial to setup strict memory containment between NUMA nodes.
- This prevents PEs from one NUMA node allocating memory on another NUMA node.
- This 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

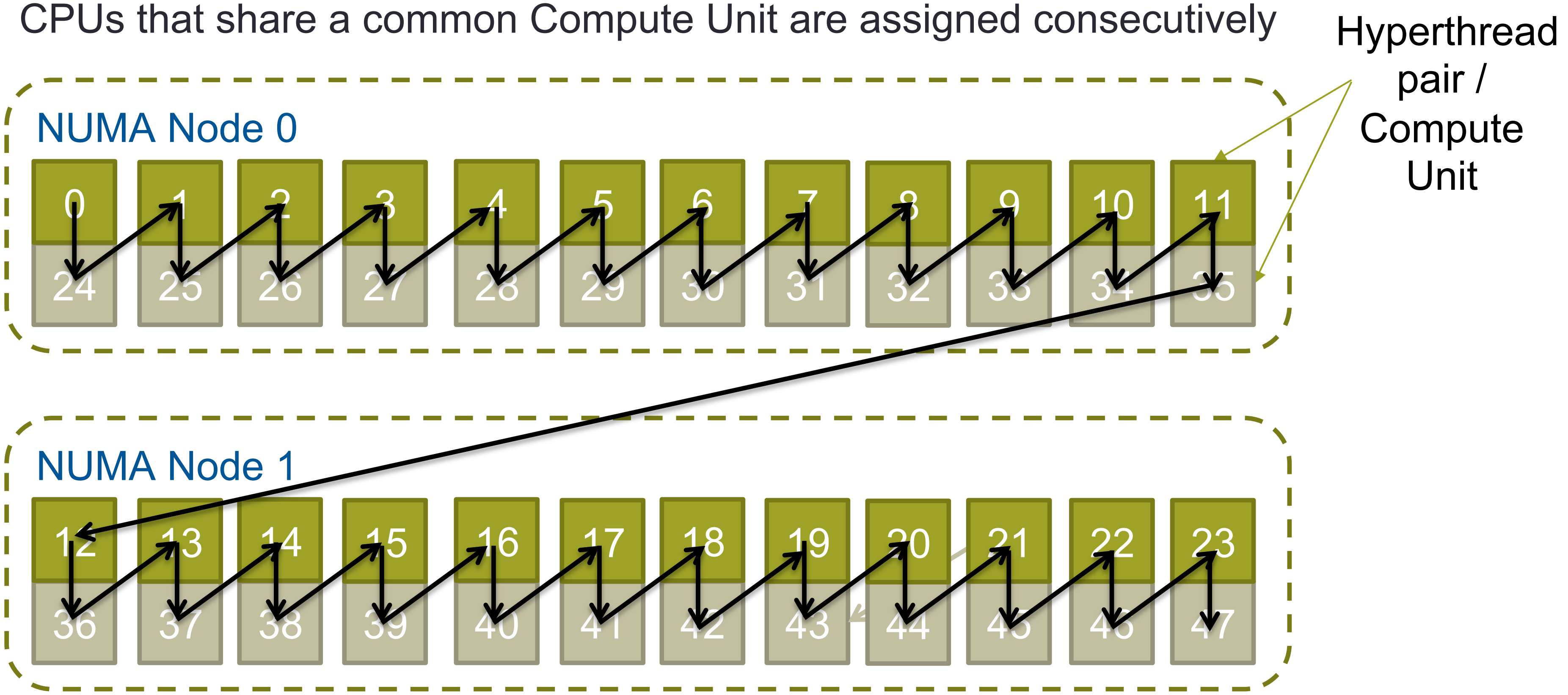
All examples up to now have assumed "`-j1`" or "Single Stream Mode".

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

Summary

- ARCHER Nodes
  - 2 x 12-core Intel Xeon Ivy-Bridge processors
  - 64 GB Memory
- General multi-core issues same as any other general HPC system around at the moment
- Hyperthreading is supported and may increase performance
  - But may not, so watch this space or try for yourselves
- On core vectorisation (AVX) needed for maximum performance
  - Generally compiler will do this but…
  - …can help the compiler and check what it's doing
- Controlling process binding can be beneficial
  - Generally, plain MPI jobs easy, but other things can be achieved