High-Performance Computing

02616

Large-scale Modelling

Recap from Week 01

- parallel architectures:
  - UMA, NUMA, cc-NUMA
- first steps in MPI:
  - “Hello World” (no communication)
  - “Hello Neighbour” (communication)
  - Ring (communication)
- everything on a single node

Topics for Week 02

- Memory bandwidth and performance
  - “NUMA in action”
- Networks
  - cluster setup
  - Infiniband and Ethernet
  - bandwidth and latency
- MPI jobs on multiple nodes
  - multi-node batch jobs
  - process placement and binding

Memory bandwidth

Single-core peak performance:

- The TPP is easy to calculate: let us assume
  - 2 GHz CPU/core
  - 4 Flops per clock cycle
  - => TPP: 8 GFlop/s (per core!)
- To obtain that peak performance, we need to be able to feed the core with the right amount of data!

Memory bandwidth

How much data is that?

- 4 floating point operations (add, mult) need 8 floating numbers => 64 bytes (double precision)
- That is 64 bytes / clock cycle or 128 GB/s
  - 128 GB is the content of more than 27 DVDs!!!
- But what is the memory bandwidth in modern machines?

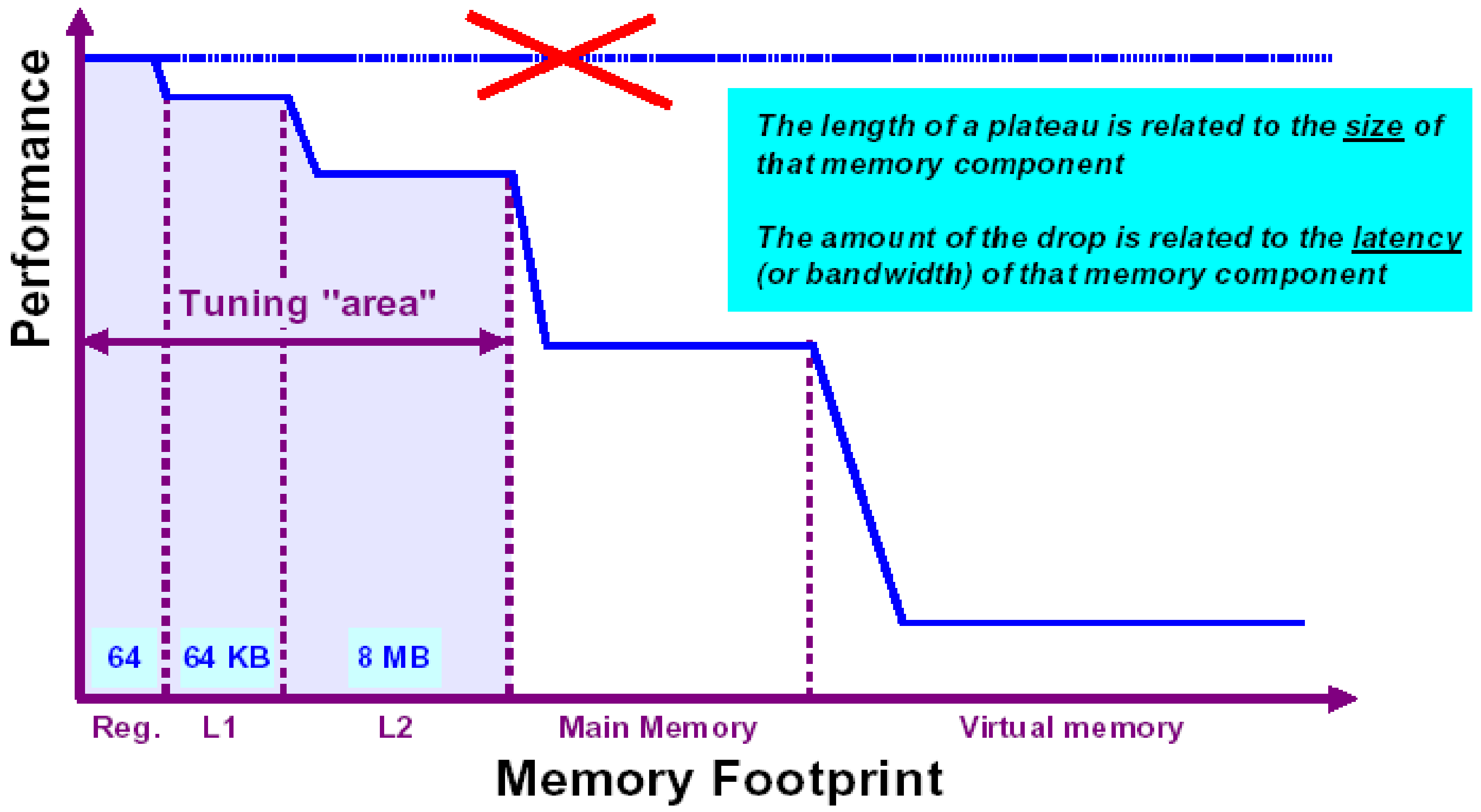
Memory bandwidth

Memory bandwidth of the Xeon E5-2650 v4 family

- equipped with DDR4-2400 DIMMS:
  - 2400 MT/s => 19.2 GB/s per memory channel
  - 76.8 GB/s maximum (all 4 channels equipped)
- with slower DDR4-1600 DIMMS:
  - max. 51.2 GB/s
- that is the bandwidth per CPU socket
  - … but each CPU has 12 cores!
  - => the memory bandwidth per core is less!!!!

Memory bandwidth

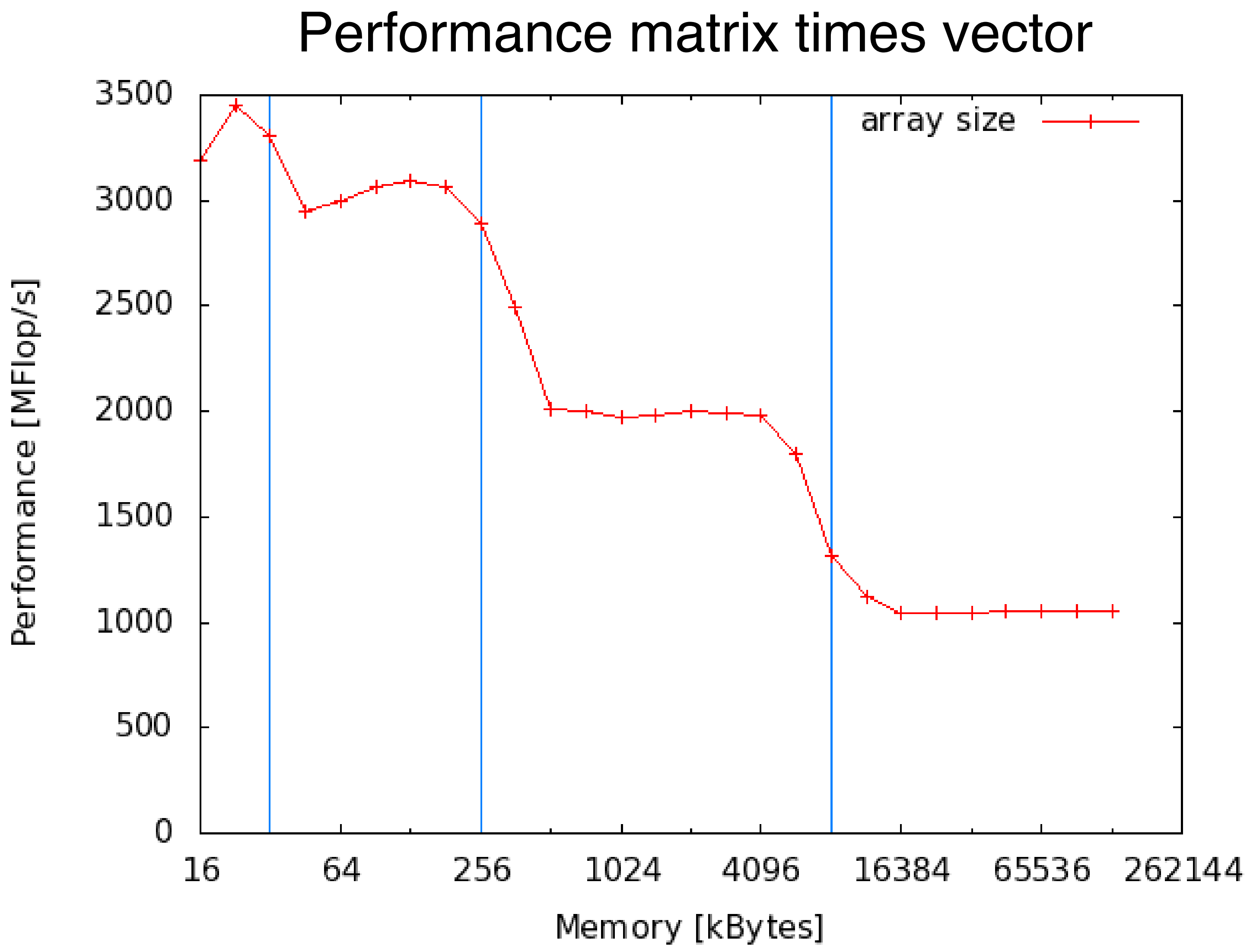
Performance staircase (single core):



The length of a plateau is related to the size of that memory component

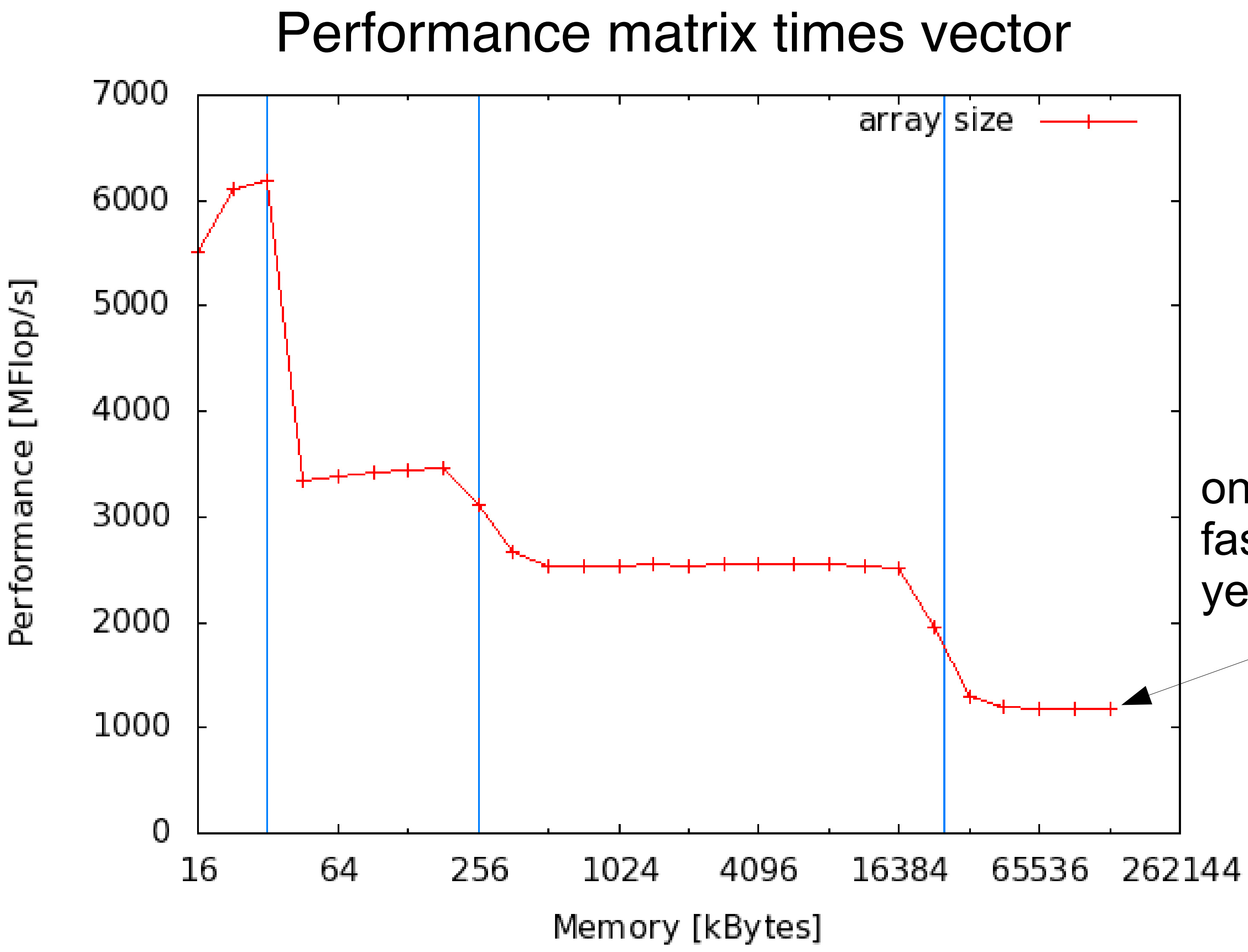
The amount of the drop is related to the latency (or bandwidth) of that memory component

Memory bandwidth



Intel Xeon X5550, 2.67 GHz, 8 MB L3 cache (release: Q1/2009)

Memory bandwidth



only ~10% faster than 5 years earlier!

Intel Xeon E5-2660v3, 2.66 GHz, 25 MB L3 (release: Q3/2014)

Memory bandwidth

Compute-bound vs memory-bound

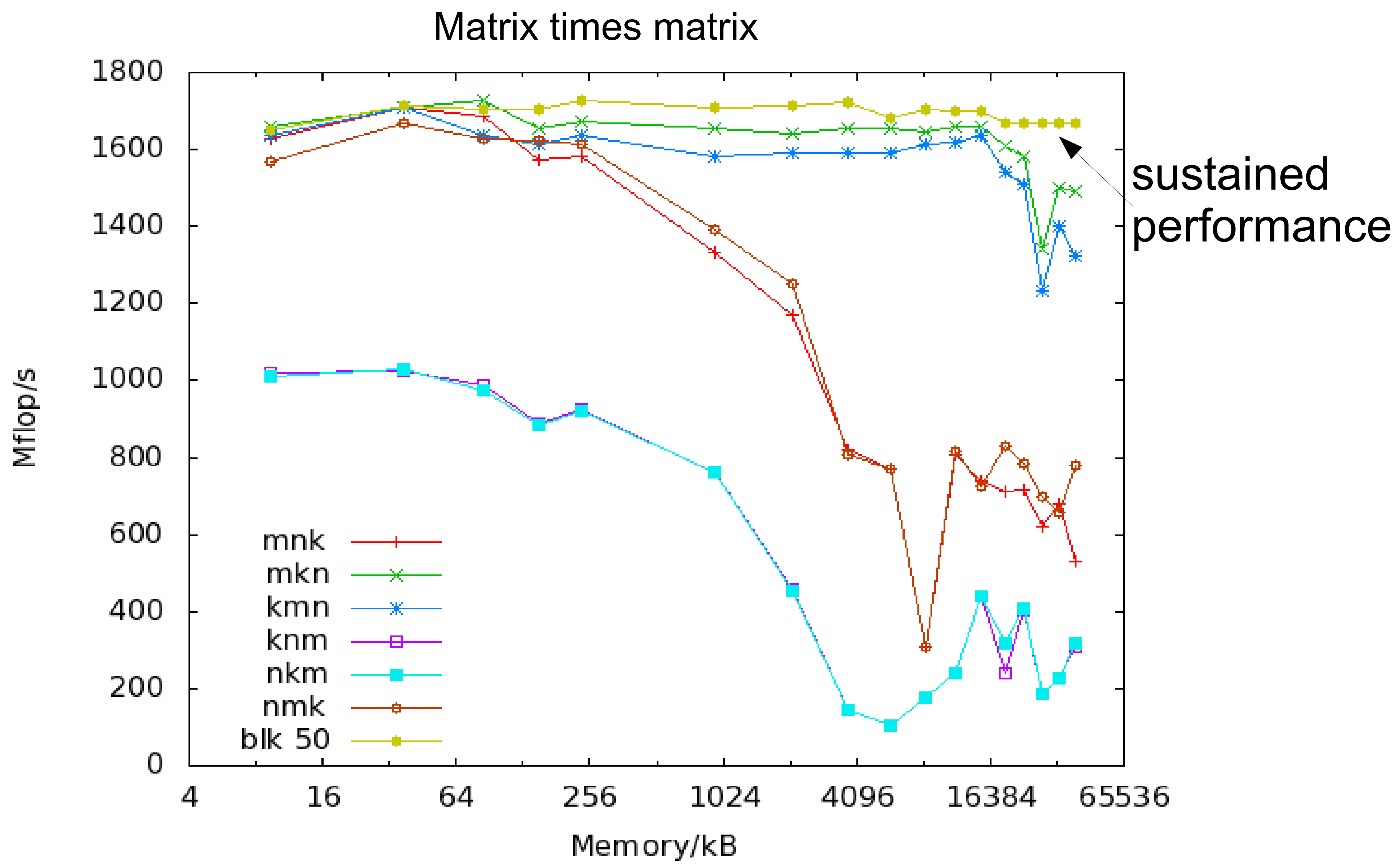
- Compute-bound:
  - the number of flops is higher than the number of mem-ops (load/store)
  - example: matrix times matrix
- Memory bound:
  - the number of mem-ops is dominating the flops
  - example: matrix times vector

Memory bandwidth

Compute-bound vs memory-bound

- but ...
- for large problems, even matrix times matrix gets dominated by the memory operations, it turns into a memory bound problem
- we know how to solve that – e.g. by blocking algorithms, to keep the problem compute bound

Memory bandwidth



Memory bandwidth

Scaling of memory-bound applications

- What happens with memory bound applications on multi-core systems?
- N cores – one CPU, e.g. N = 4
- N threads execute a memory-bound kernel
  - => all N threads fight for the same mem-bw
  - => this is a performance bottleneck
  - => this has a negative effect on scaling (speed-up)

Memory bandwidth

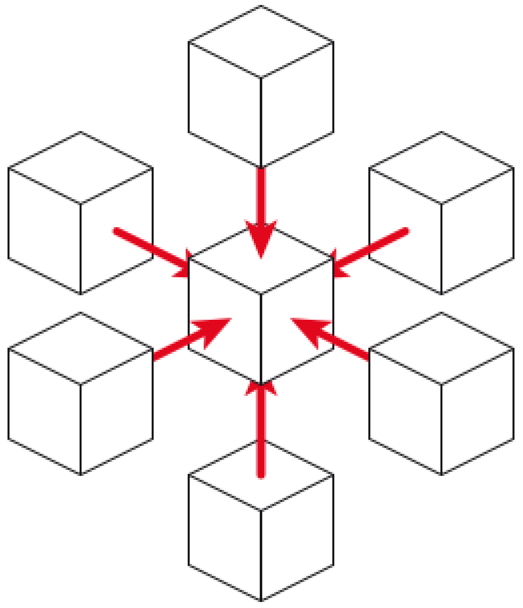
Scaling of memory-bound applications (cont'd)

- this effect is visible for all parallel applications, regardless of the programming model, i.e.
  - distributed applications (e.g. MPI)
  - shared-memory applications (e.g. OpenMP)
- adding more cores to the CPU makes the situation even worse!
- adding more sockets helps:
  - increased memory bandwidth
  - the overall performance is bounded by the max. memory bandwidth (saturation)

Memory bandwidth

Example (from the 02614 course):

- 3D Poisson problem: N^3 grid points
- 7 point stencil
- 8 mem-ops per stencil update
- memory bound problem
- data: double precision (64 bit)
- N = 1000
- memory usage: 22.5 GB
- OpenMP parallel implementation



Memory bandwidth

Example (from the 02614 course):

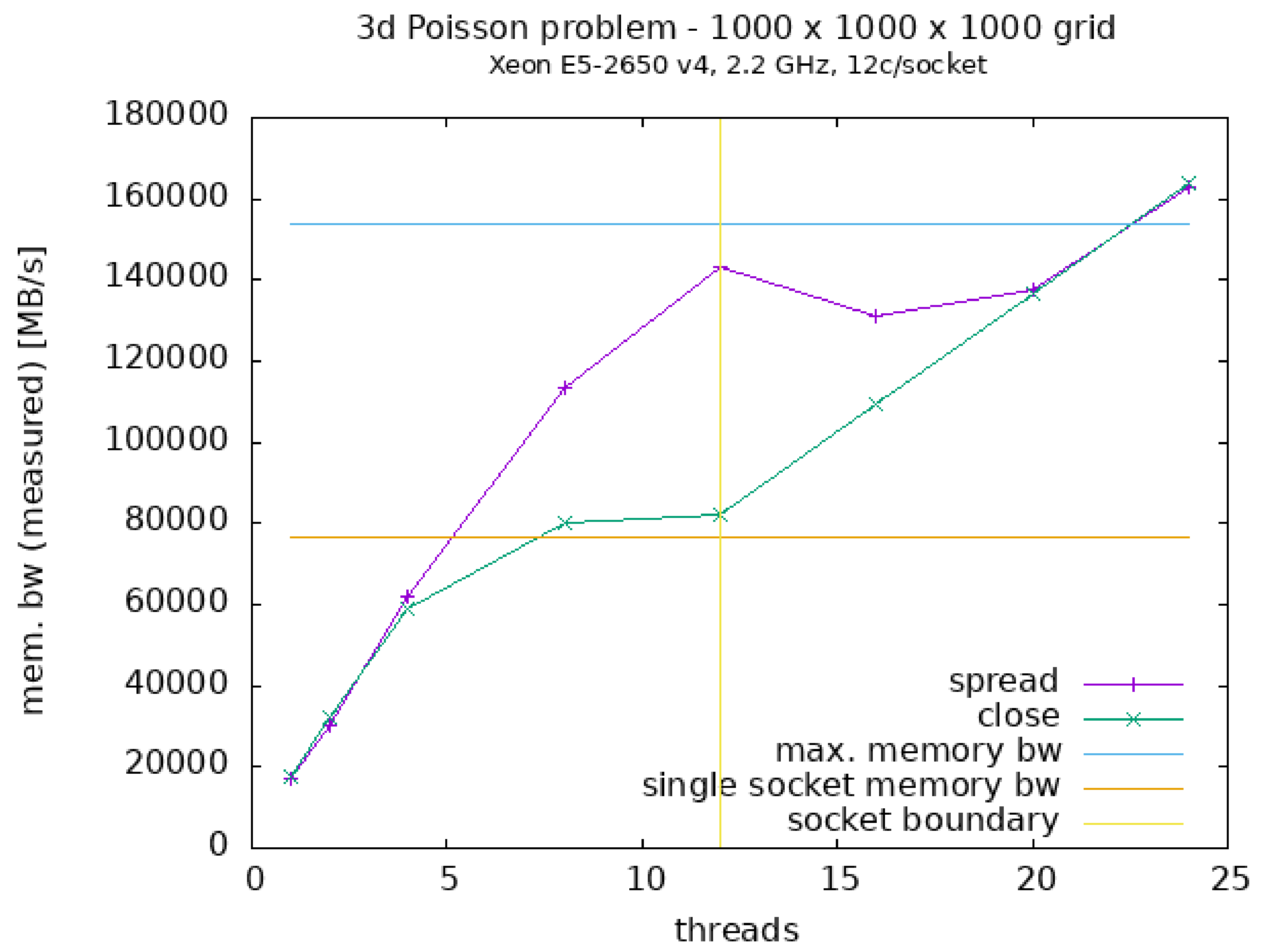
- 3D Poisson problem: 1000^3 grid points
- Hardware:
  - 2x Intel XeonE5-2650 v4, 2.2 GHz
  - L3 size: 30MB
  - 256 GB RAM (128 GB per socket)
  - single core TPP: 35.2 Gflop/s
  - 12 cores per socket
  - memory bandwidth per socket: 76.8 GB/s

Memory bandwidth

How did we do?

- ❑ we can measure the number of stencil updates (lup) per second: Mlup/s
- ❑ each lup involves 8 memory operations:
  - ❑ 7 loads and 1 store
  - ❑ uncertainty: loads and stores can overlap
- ❑ our data type is double: 8 bytes
  - ❑ => 64 bytes per lup
- ❑ measured memory bandwidth:
  - ❑ MB/s = Mlup/s * 64 bytes

Memory bandwidth



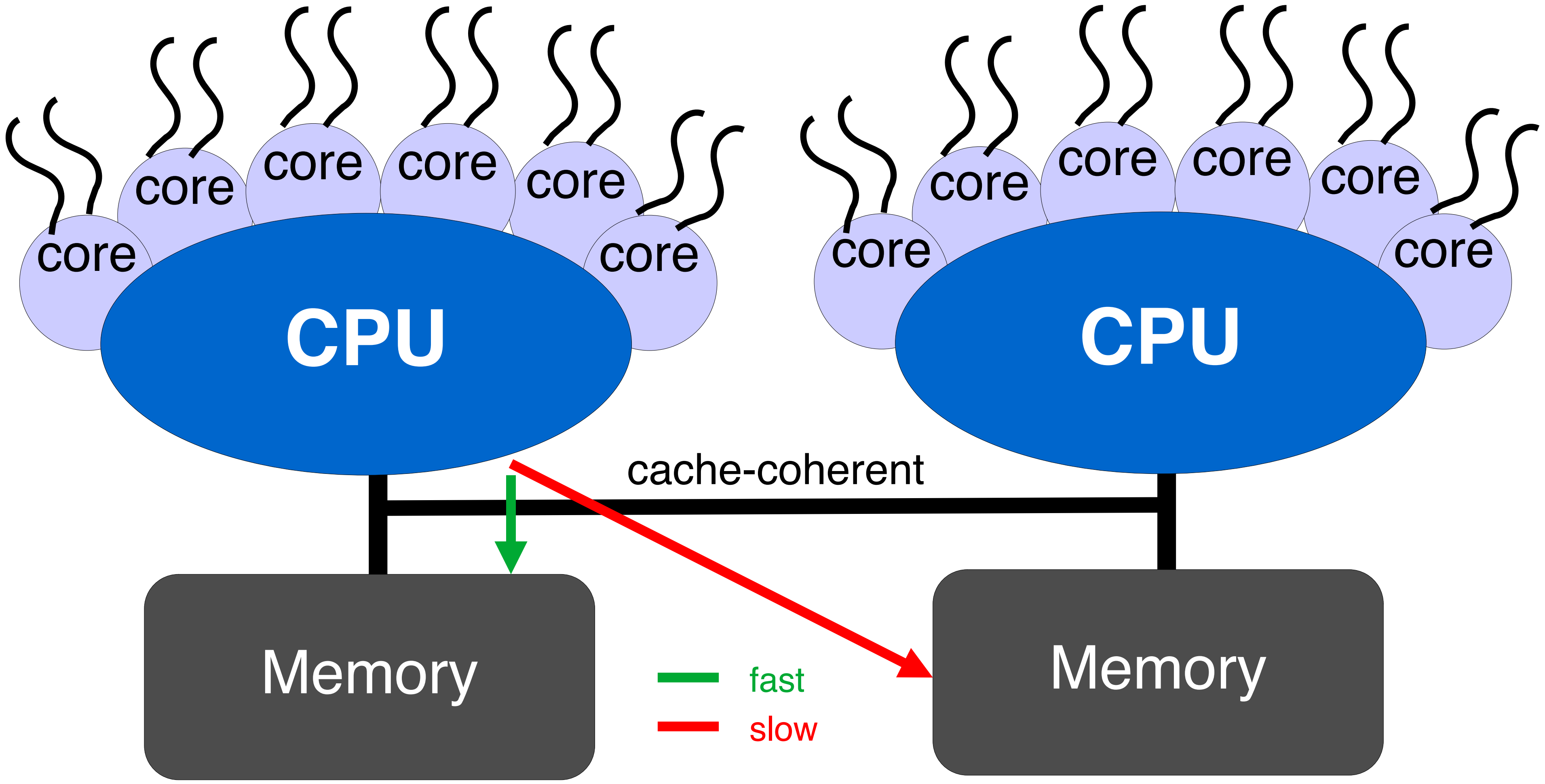
Memory bandwidth

- Lessons learned:
  - using a single socket (close distribution), we are already limited by the socket memory bandwidth when using 8 threads on the 12 cores
  - spreading the threads over both sockets is beneficial for up to 12 threads, i.e. 6 threads per socket
  - for larger number of threads, e.g. 20 and 24 threads (the full node), the two distributions approach each other.
  - similar behaviour can be observed when using an MPI implementation of the problem.

Networks

Cluster of multi-socket machines

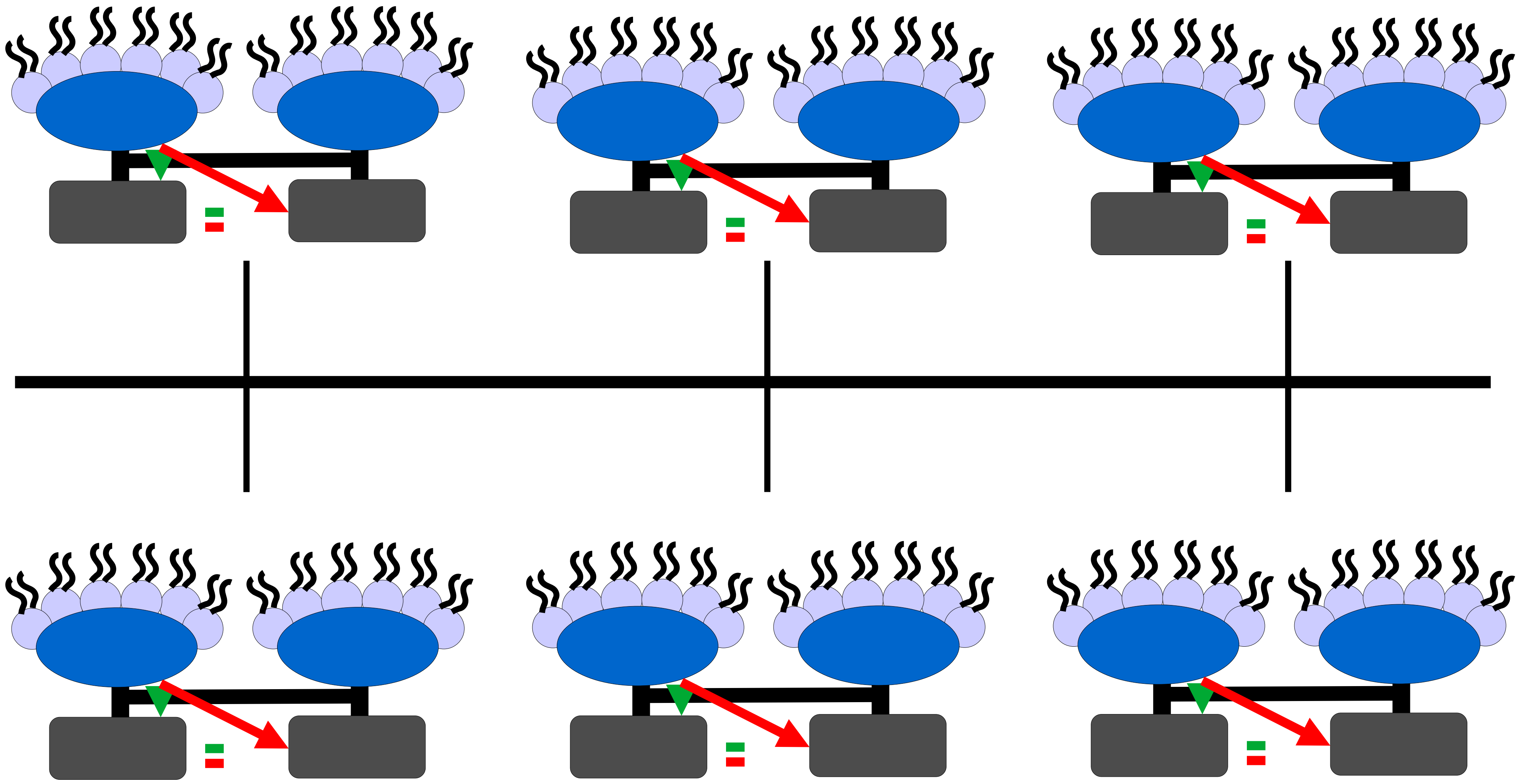
A typical multi-core server



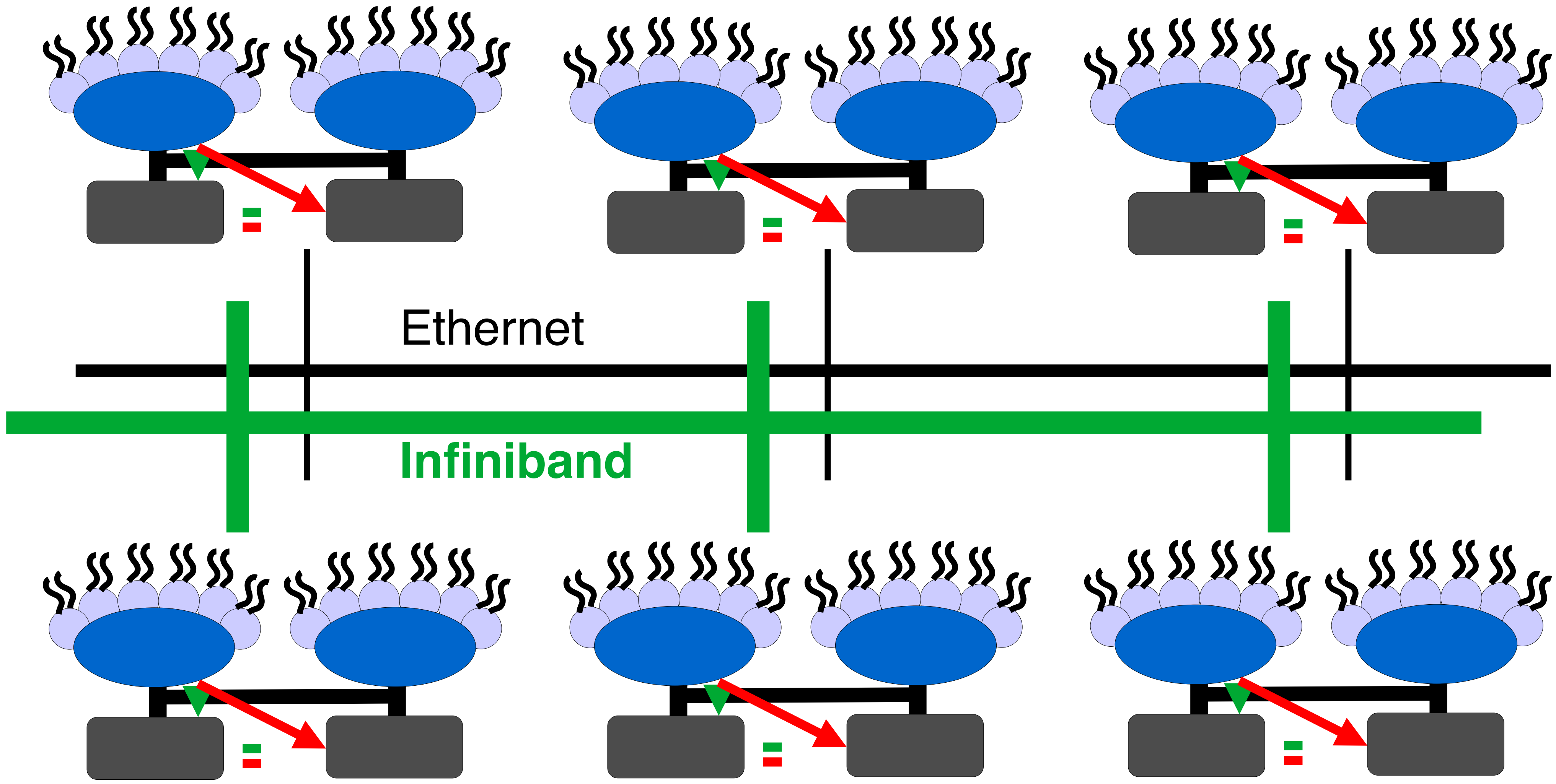
a 2-socket, 12-core, 24-(hyper-)threads server ⇒ 24 logical CPUs

Note: we do not use hyperthreading in our setup!

Cluster with a single network



Cluster with two networks



Why a fast network?

- for HPC applications, we want to have
  - a high network bandwidth
  - low latency
  - => minimizing the communication overhead
- this can be achieved by networks like Infiniband (IB), OmniPath (OPA), etc
- other advantages of IB
  - RDMA: remote direct memory access – doesn't involve the CPU when accessing memory on a remote node
  - GPU Direct: RDMA between a local and remote GPU

Why a fast network?

Infiniband specifications:

- there are several generations of IB
- the latest generations provide
  - up to 400 Gb/s
  - 3-5 microseconds latency
- in our setup:
  - FDR IB: up 56 Gb/s
  - compare to Ethernet: either 1 Gb/s or 10 Gb/s
  - today's lab: measure the difference in bandwidth and latency between IB and Ethernet

Fun fact

What is the memory bandwidth of our brain?

- It's really slow: 10 bit/s!
- Read more about it
  - "The unbearable slowness of being: Why do we live at 10 bits/s?"

The “real” world

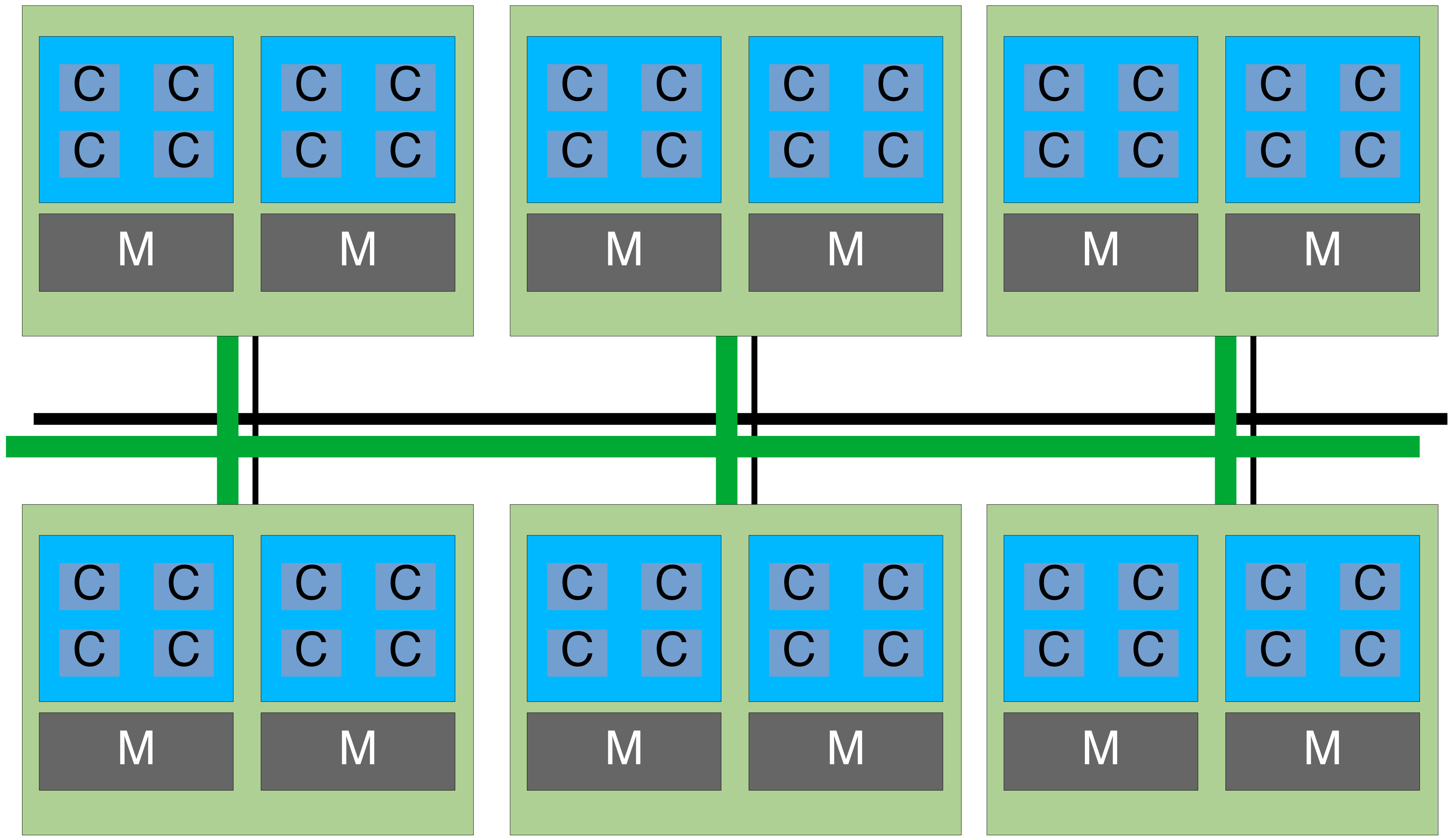
Putting it all together

The “real” world

What we have learned so far:

- The limiting factor in many applications is bandwidth:
  - memory bandwidth in a single node
  - network bandwidth on a network of nodes
- Now let us put those pieces together:
  - a cluster of multi-socket (cc-NUMA) systems,
  - connected by both Infiniband and Ethernet

The “real” world



The “real” world

- run an MPI application
  - with N ranks (single-threaded)
  - on M nodes (dual-socket)
  - each node has 2*P cores (P cores/socket)
- we will have 2*P*M locations to place our N ranks on ($N \leq 2*P*M$)
- How does the placement influence the performance of our application?
- How can we control it?
  - interplay between MPI and scheduler (LSF)

The “real” world

Specific example:

- N = 8 ranks
- M = 4 nodes
- P = 4 cores/socket (8 cores per node)
- 32 possible locations for the 8 ranks

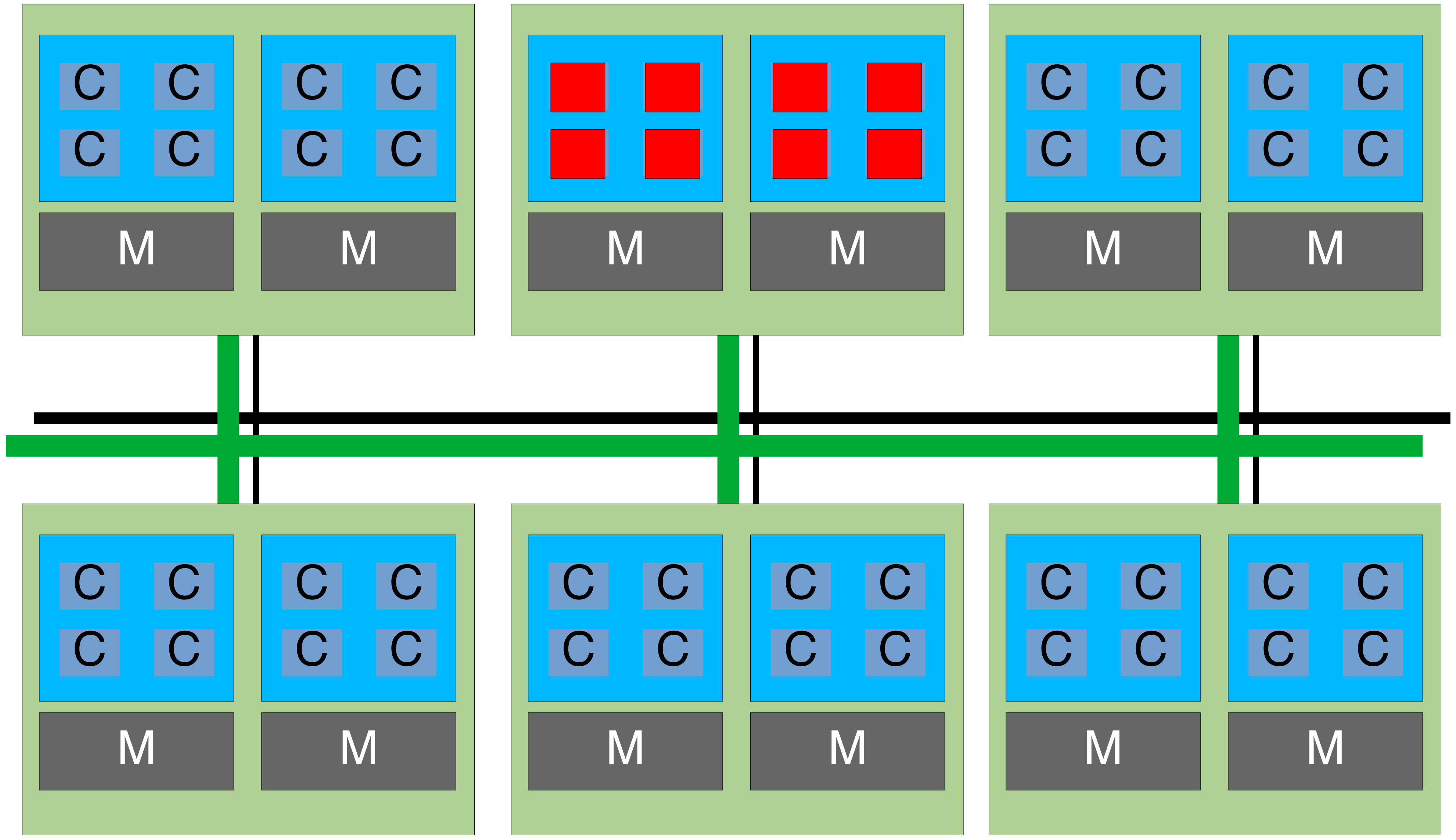
The “real” world

Scenario 1:

- all 8 ranks on a single node

```
#BSUB -J MPIapp
#BSUB -o MPIapp_%J.out
#BSUB -n 8
#BSUB -R "span[hosts=1]"

module load mpi/<version>

mpirun ./myapp
```

The "real" world



The “real” world

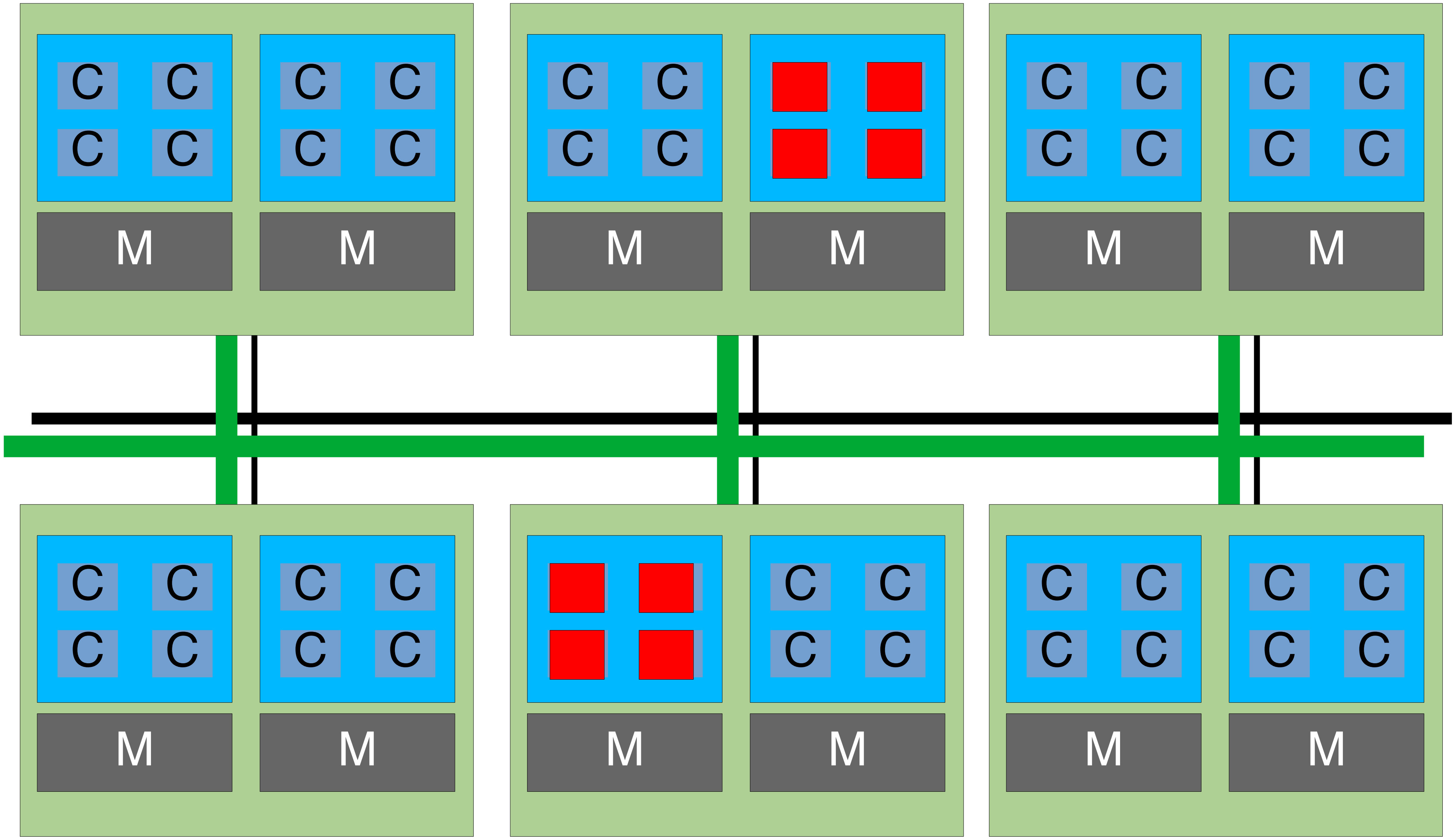
Scenario 2:

- use two nodes, 4 ranks per node

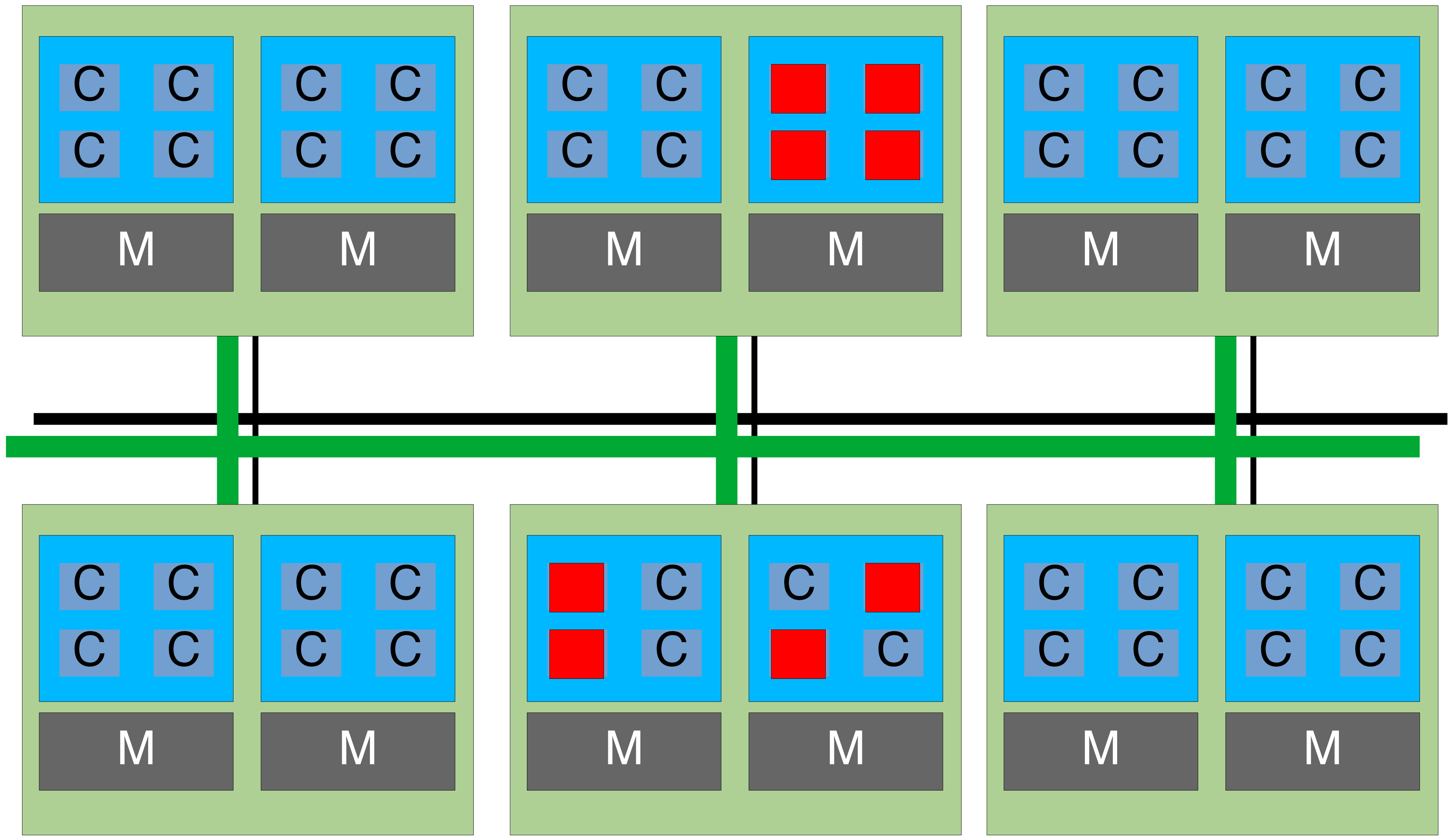
```
#BSUB -J MPIapp
#BSUB -o MPIapp_%J.out
#BSUB -n 8
#BSUB -R "span[ptile=4]"

module load mpi/<version>

mpirun ./myapp
```

- What do we get?

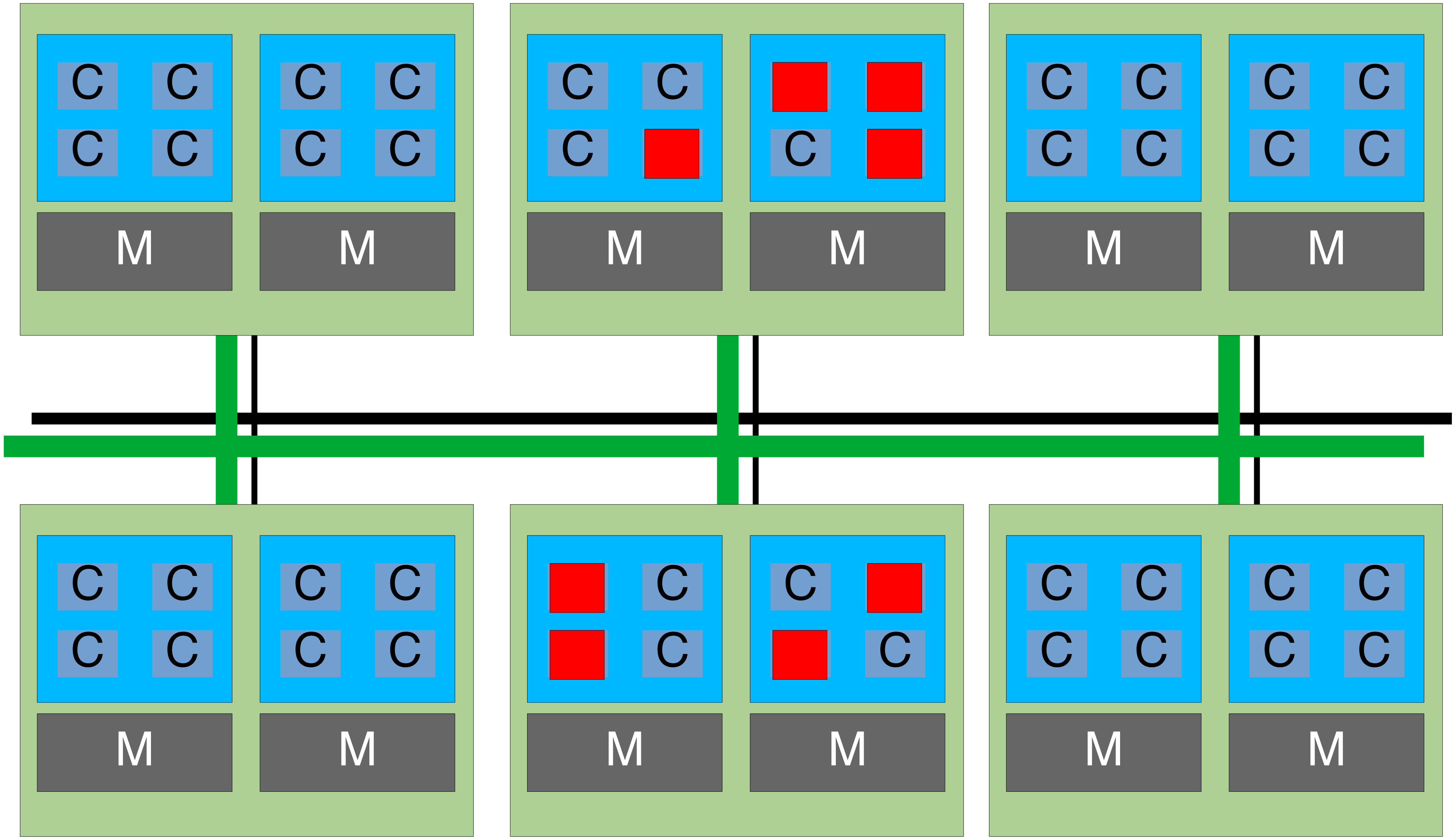
The “real” world



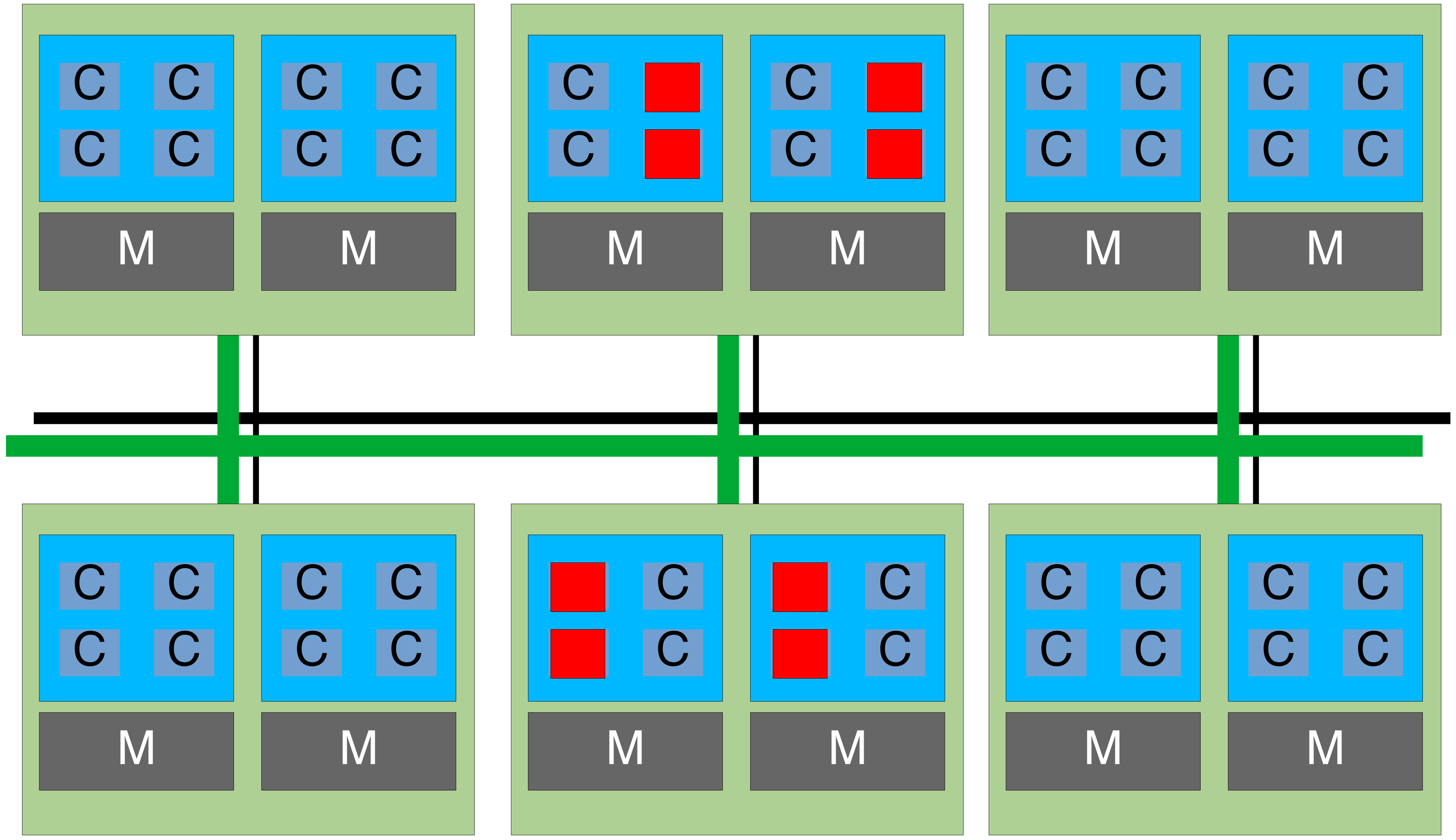
The “real” world



The “real” world



The “real” world



The “real” world

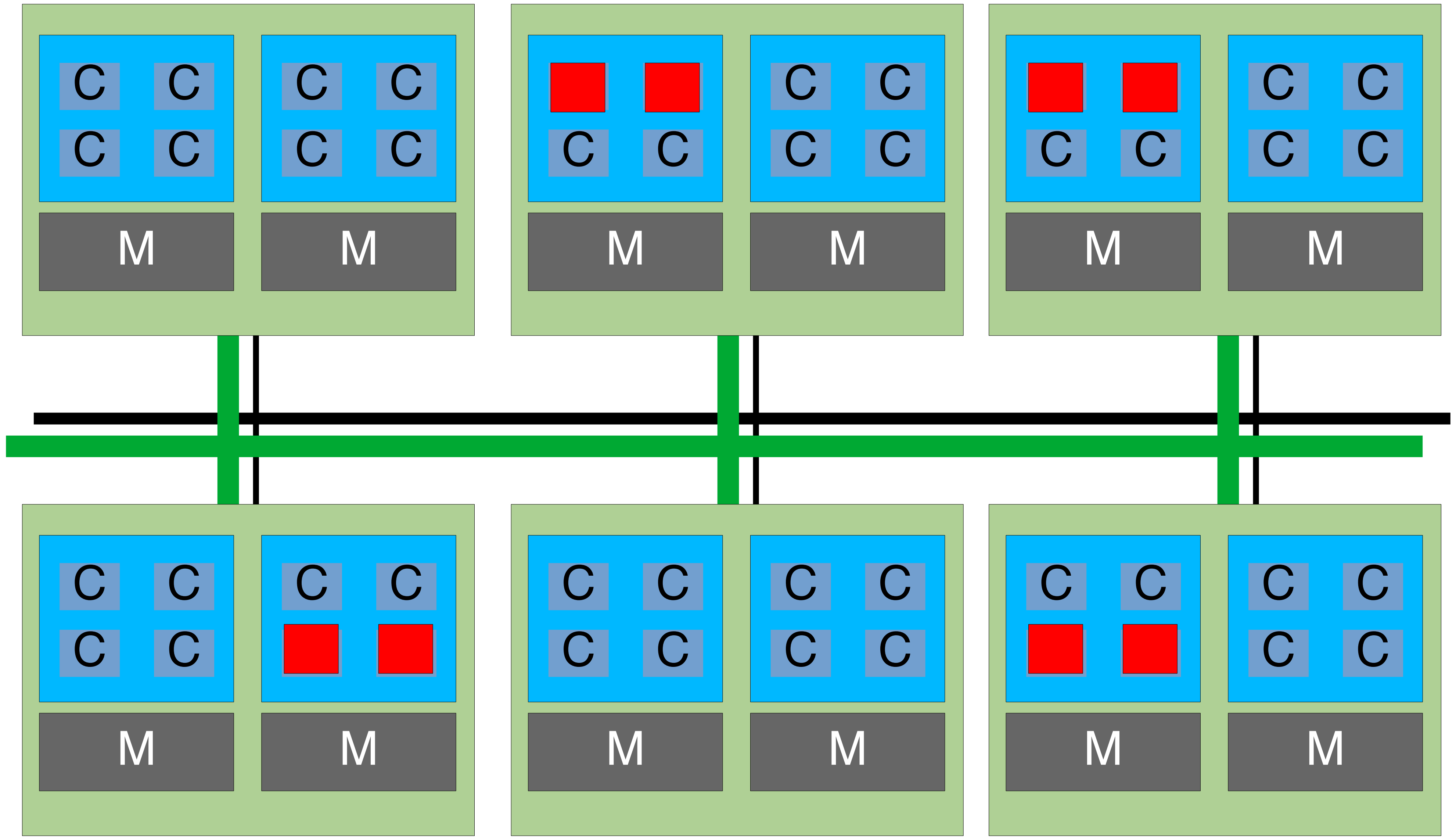
Scenario 3:

- use 4 nodes, 2 ranks per node

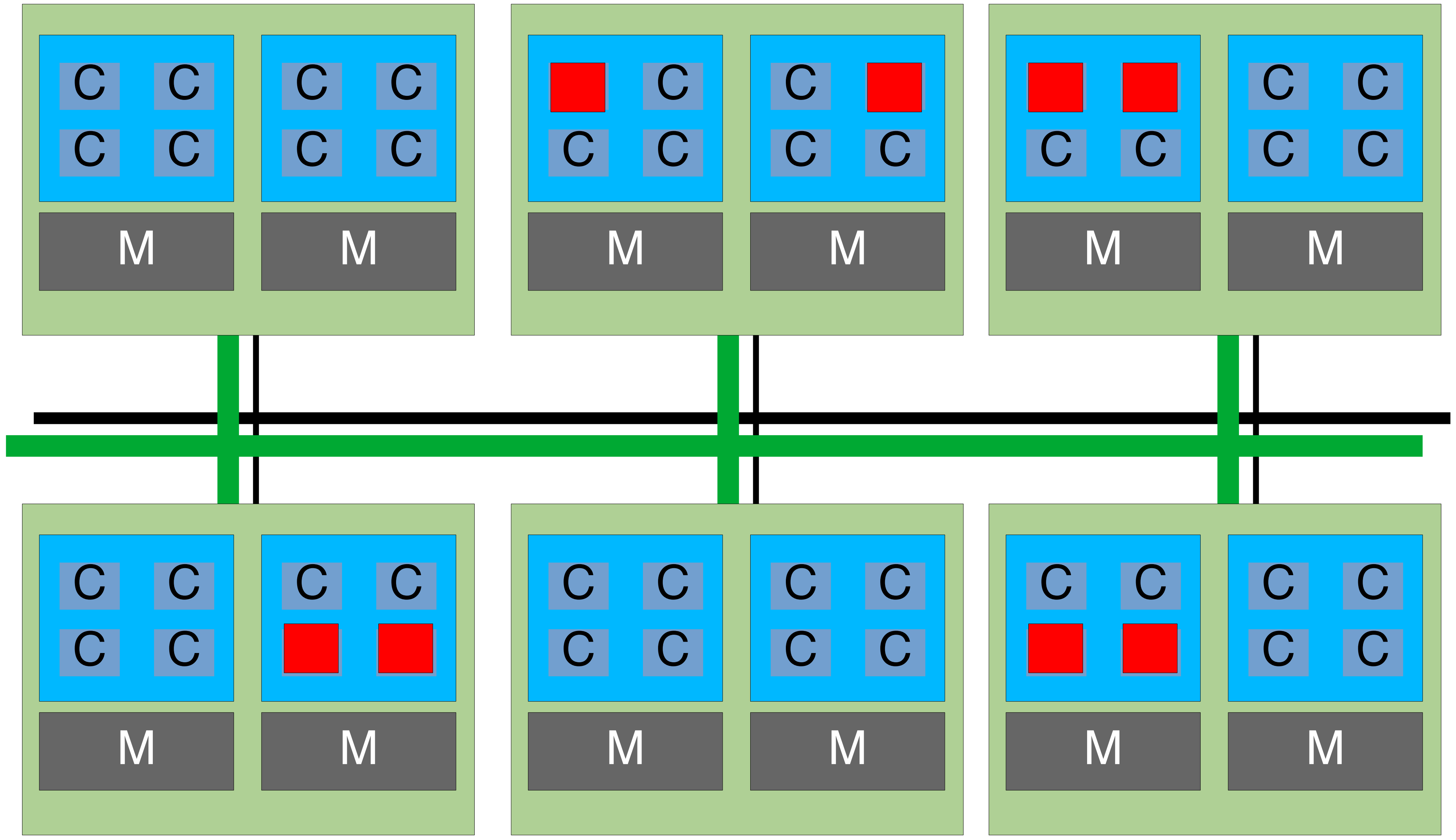
```
#BSUB -J MPIapp
#BSUB -o MPIapp_%J.out
#BSUB -n 8
#BSUB -R "span[ptile=2]"

module load mpi/<version>

mpirun ./myapp
```

- What do we get?

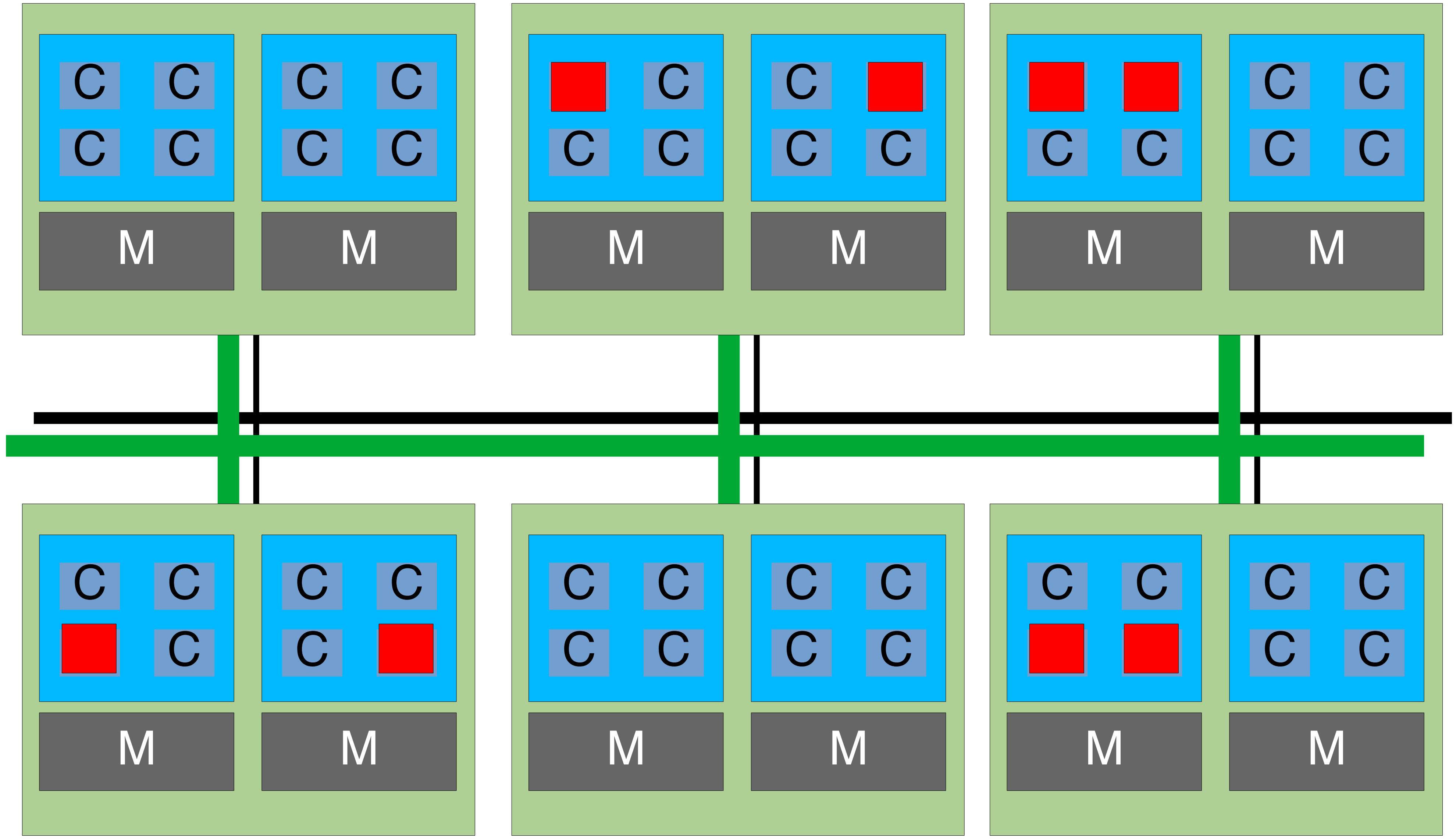
The “real” world



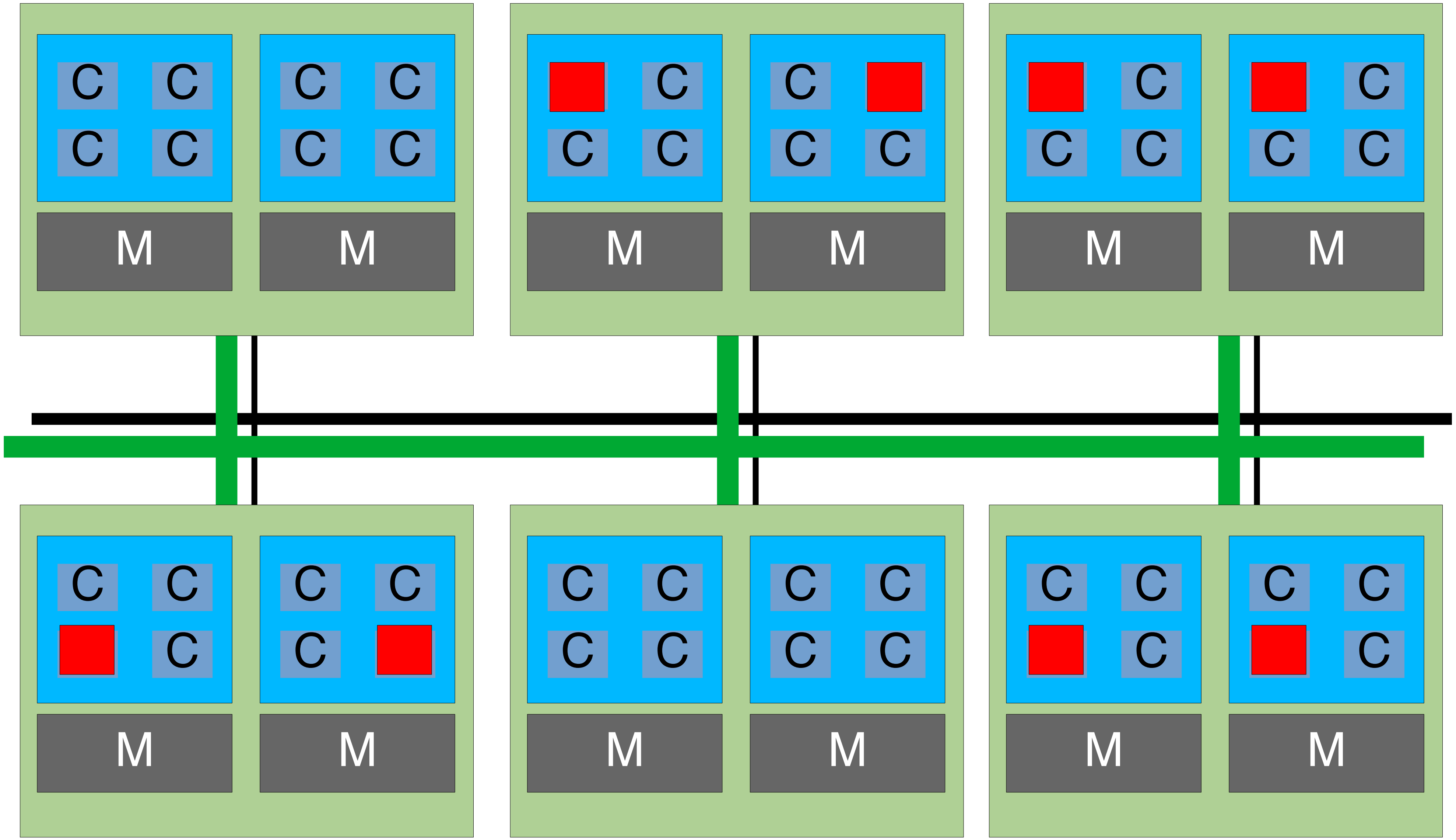
The “real” world



The “real” world



The “real” world



The “real” world

- If our MPI application needs a lot of memory bandwidth, then we want to distribute the ranks over as many CPU sockets as possible.
- Here, the OpenMPI runtime options for mapping can help
- Syntax:
  - mpirun –map-by ppr:n:<resource>
  - ppr stands for “process(es) per resource”
  - <resource> can be package (a CPU socket), core, …
  - check the documentation for more details

The “real” world

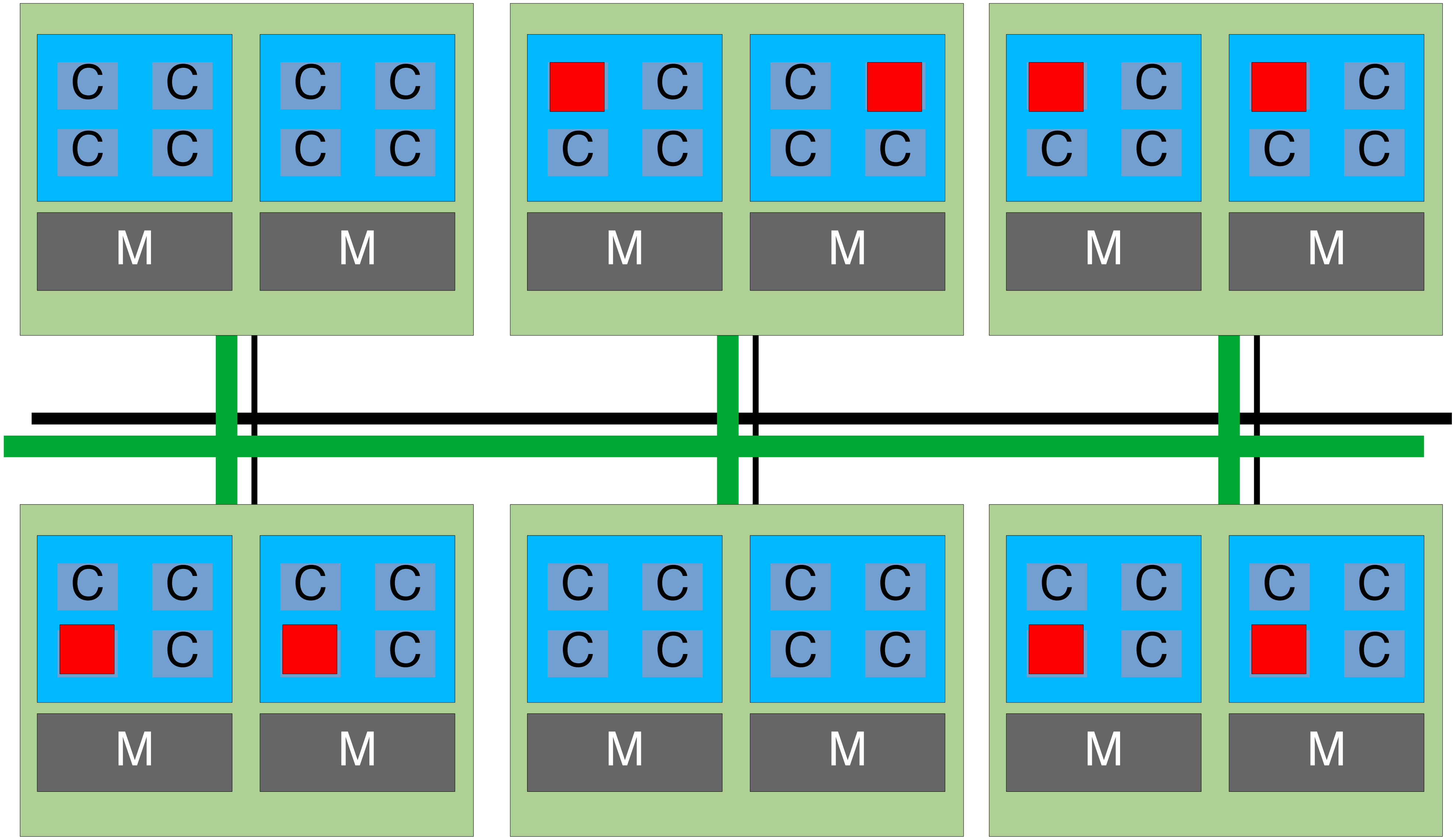
Scenario 3 – with mapping

- use 4 (full) nodes, 2 ranks per node

```
#BSUB -J MPIapp
#BSUB -o MPIapp_%J.out
#BSUB -n 32
#BSUB -R "span[ptile=8]"

module load mpi/<version>

mpirun -np 8 -map-by ppr:1:package ./myapp
```

- Then we get something like … on the next slide

The “real” world



The “real” world

- The solution above gives us the maximum bandwidth for our problem
- … for the price of allocating more resources than we need!
- There are other ways to implement this, e.g. using the affinity options of LSF, both with and without additional MPI runtime options
  - … helps to save resources
  - … but that’s beyond the scope of today!
- Today’s lab: measure runtime of such a placement scenario