Load balance & rank placement

Motivation for load imbalance analysis

- **Increasing system, software and architecture complexity**
  - Current trend in high end computing is to have systems with tens of thousands of processors
    - This is being accentuated with multi-core processors
- **Applications have to be very well balanced In order to perform at scale on these MPP systems**
  - Efficient application scaling includes a balanced use of requested computing resources
- **Desire to minimize computing resource "waste"**
  - Identify slower paths through code
  - Identify inefficient "stalls" within an application

Example load distribution

Imbalance time

- **Metric based on execution time**
- **It is dependent on the type of activity:**
  - User functions

    Imbalance time = Maximum time – Average time
  - Synchronization (Collective communication and barriers)

    Imbalance time = Average time – Minimum time
- **Identifies computational code regions and synchronization calls that could benefit most from load balance optimization**
- **Estimates how much overall program time could be saved if corresponding section of code had a perfect balance**
  - Represents upper bound on “potential savings”
  - Assumes other processes are waiting, not doing useful work while slowest member finishes

Imbalance %

$$\text{Imbalance\%} = 100 \times \frac{\text{Imbalance time}}{\text{Max Time}} \times \frac{N}{N - 1}$$

- Represents % of resources available for parallelism that is "wasted"
- Corresponds to % of time that rest of team is not engaged in useful work on the given function
- Perfectly balanced code segment has imbalance of 0%
- Serial code segment has imbalance of 100%

MPI sync time

- Measure load imbalance in programs instrumented to trace MPI functions to determine if MPI ranks arrive at collectives together
- Separates potential load imbalance from data transfer
- Sync times reported by default if MPI functions traced
- If desired, `PAT_RT_MPI_SYNC=0` deactivates this feature

Causes and hints

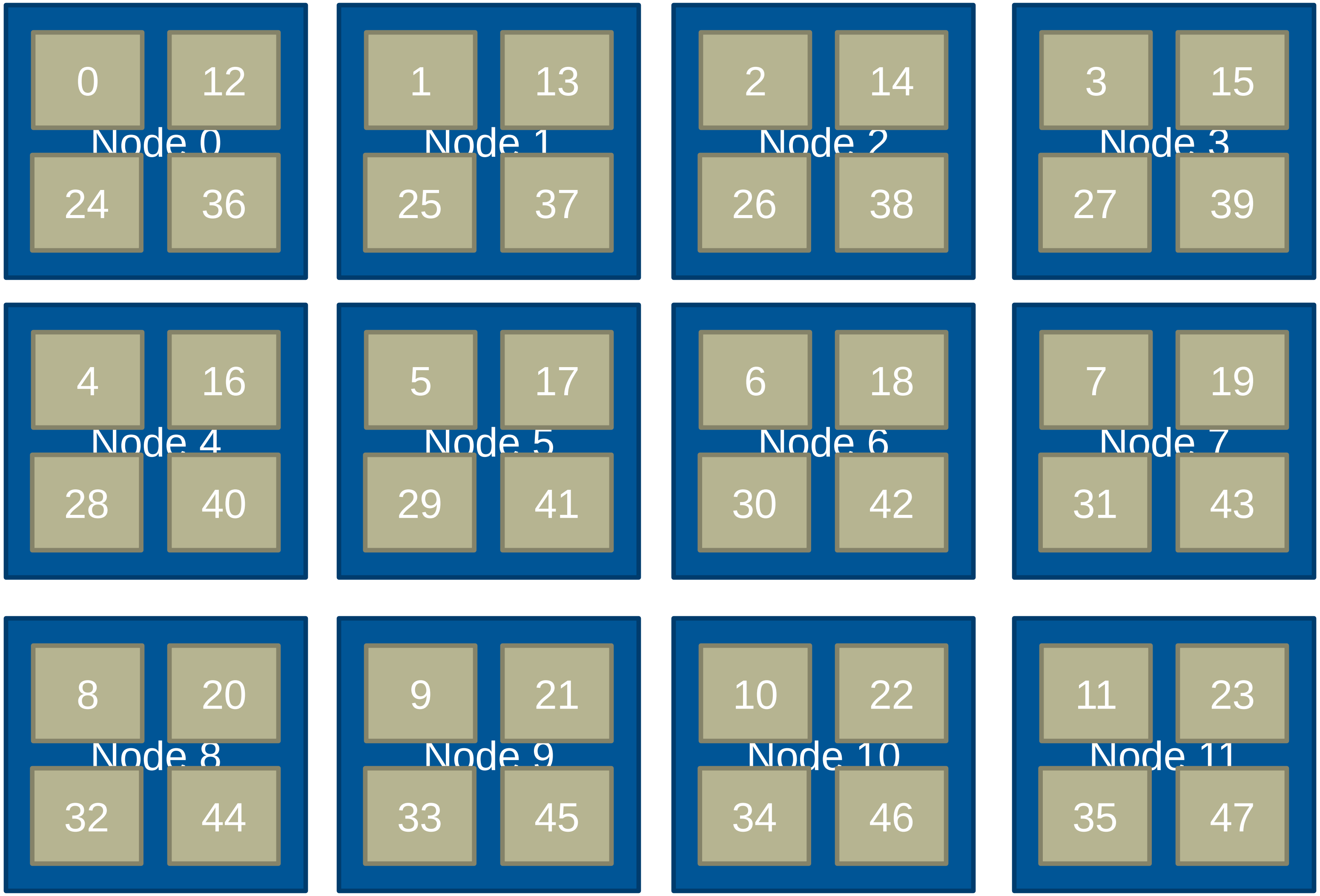
- **Need CrayPAT reports: What is causing the load imbalance?**

- **Computation**
  - Is decomposition appropriate?
  - Would reordering ranks help?
- **Communication**
  - Is decomposition appropriate?
  - Would reordering ranks help?
  - Are receives pre-posted?
  - Any All-to-1 communication?
- **I/O – synchronous single-writer I/O will cause significant load imbalance already with a couple of MPI tasks**

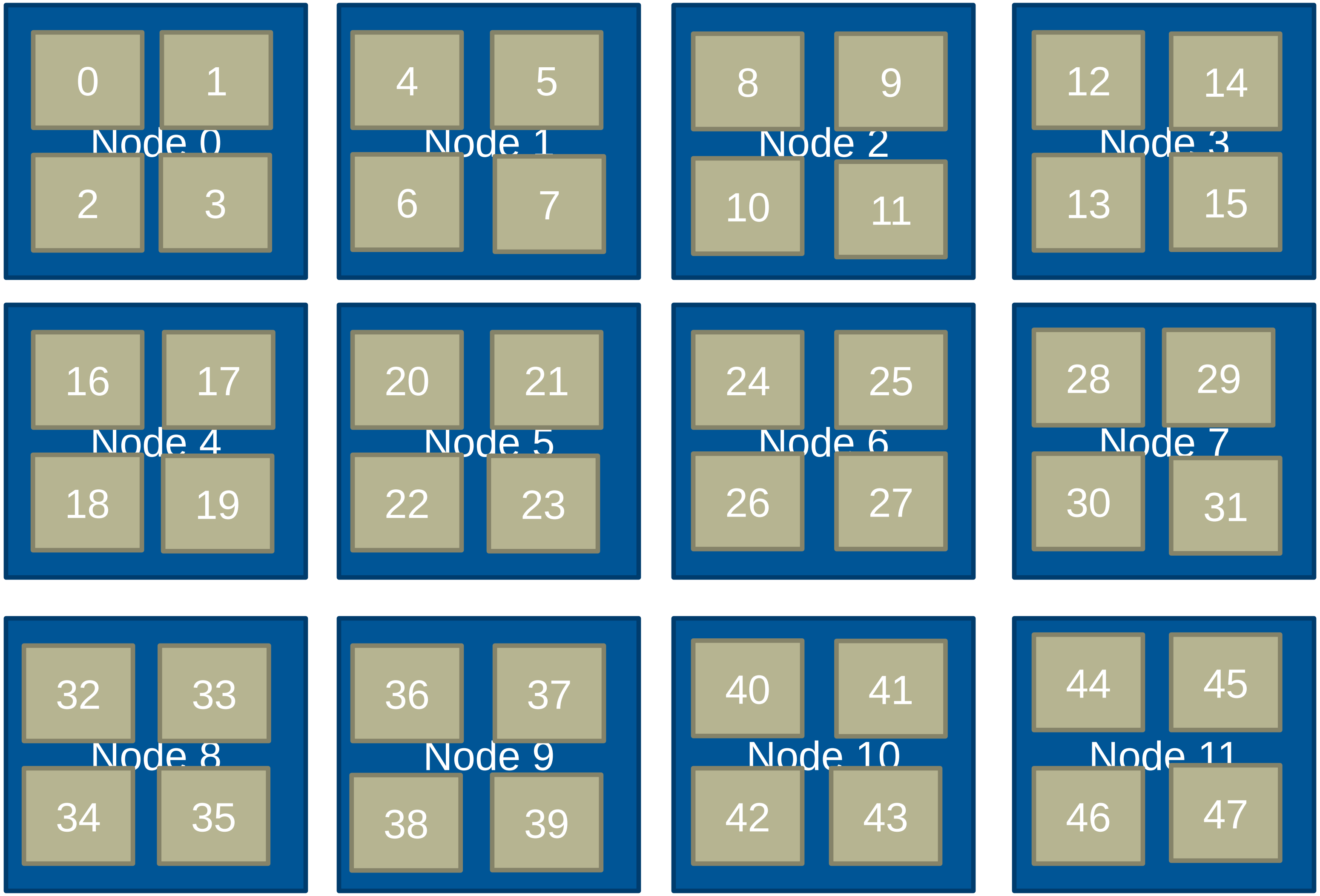
Rank Placement

- **The default ordering can be changed using the following environment variable:**
  - `MPICH_RANK_REORDER_METHOD=n`
- **These are the different values that you can set it to:**
  - **0**: Round-robin placement – Sequential ranks are placed on the next node in the list. Placement starts over with the first node upon reaching the end of the list.
  - **1**: (DEFAULT) SMP-style placement – Sequential ranks fill up each node before moving to the next.
  - **2**: Folded rank placement – Similar to round-robin placement except that each pass over the node list is in the opposite direction of the previous pass.
  - **3**: Custom ordering. The ordering is specified in a file named `MPICH_RANK_ORDER`.

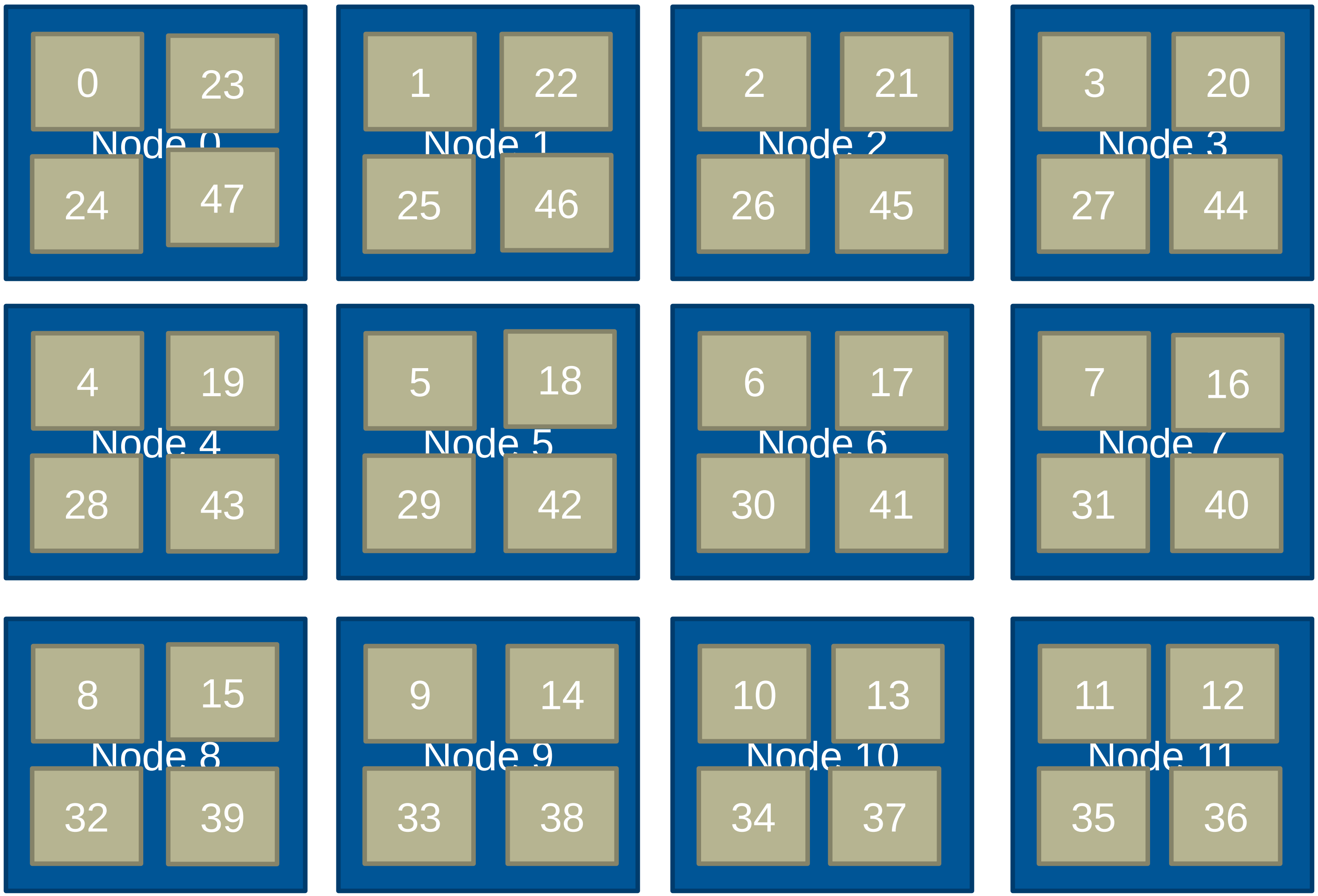
0: Round Robin Placement



1: SMP Placement (default)



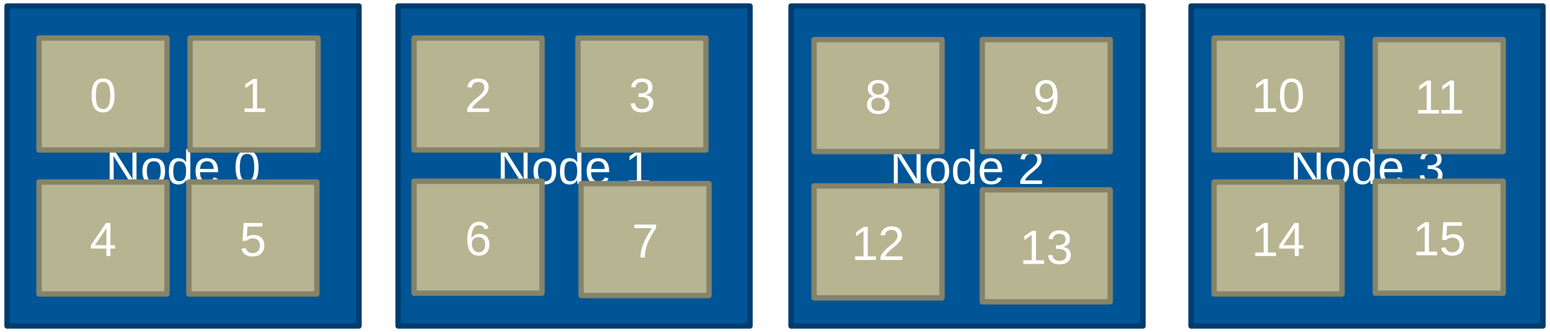
2: Folded Placement



Rank Placement

- **When is this useful?**
  - Point-to-point communication consumes a significant fraction of program time and a load imbalance detected
  - Also shown to help for collectives (all-to-all) on sub-communicators
  - To spread out IO across nodes

3: Custom Example



`MPICH_RANK_ORDER`

```
0,1,4,5,2,3,6-9,12,13,10,11,14,15
```

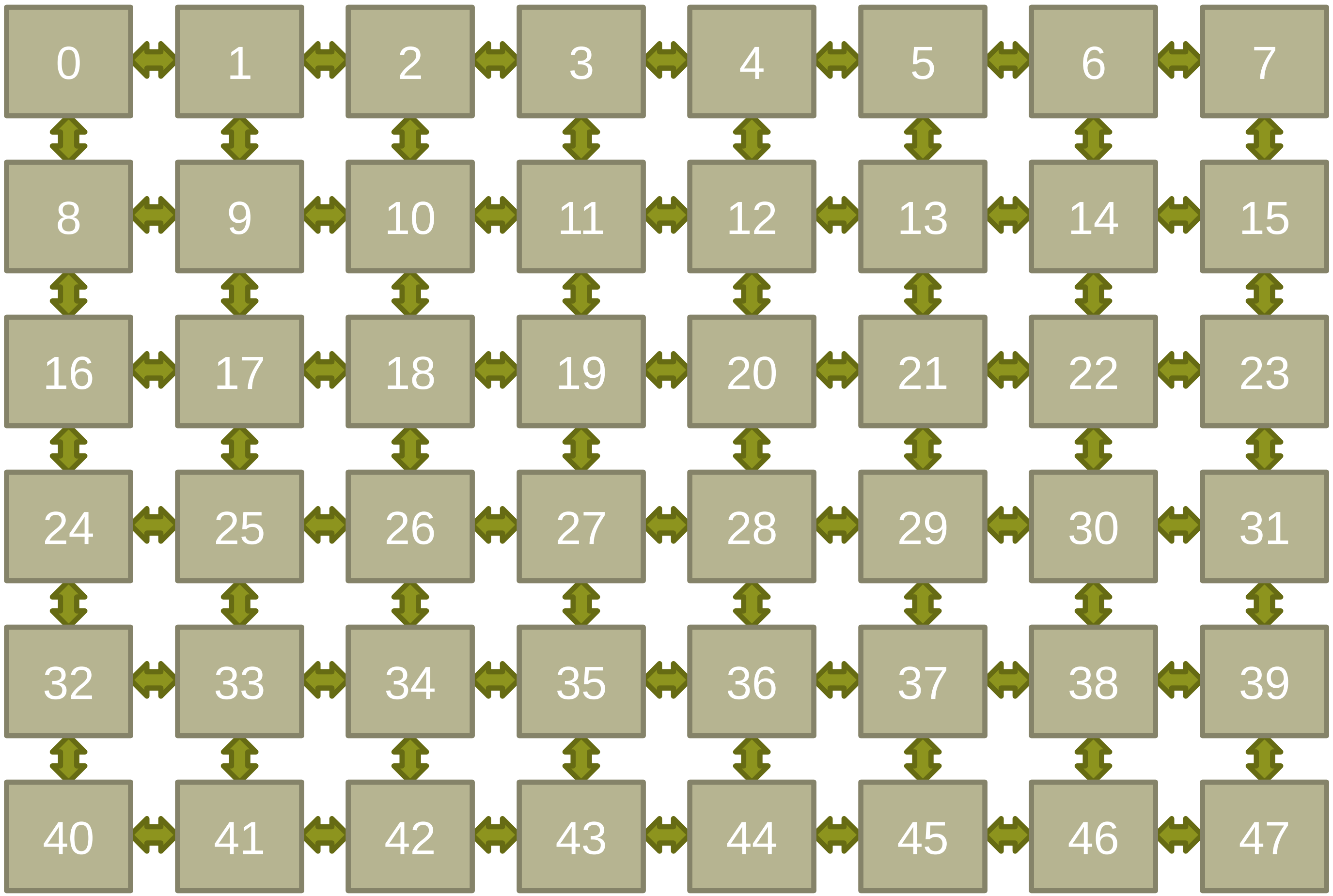
When `MPICH_RANK_REORDER=3` is set at runtime the MPI environment will read the `MPICH_RANK_ORDER` file in the current working directory and assign ranks accordingly.

`MPICH_RANK_ORDER` is a file containing a comma separated ordered list of ranges and individual rank assignments. All ranks should be included only once.

Rank reordering

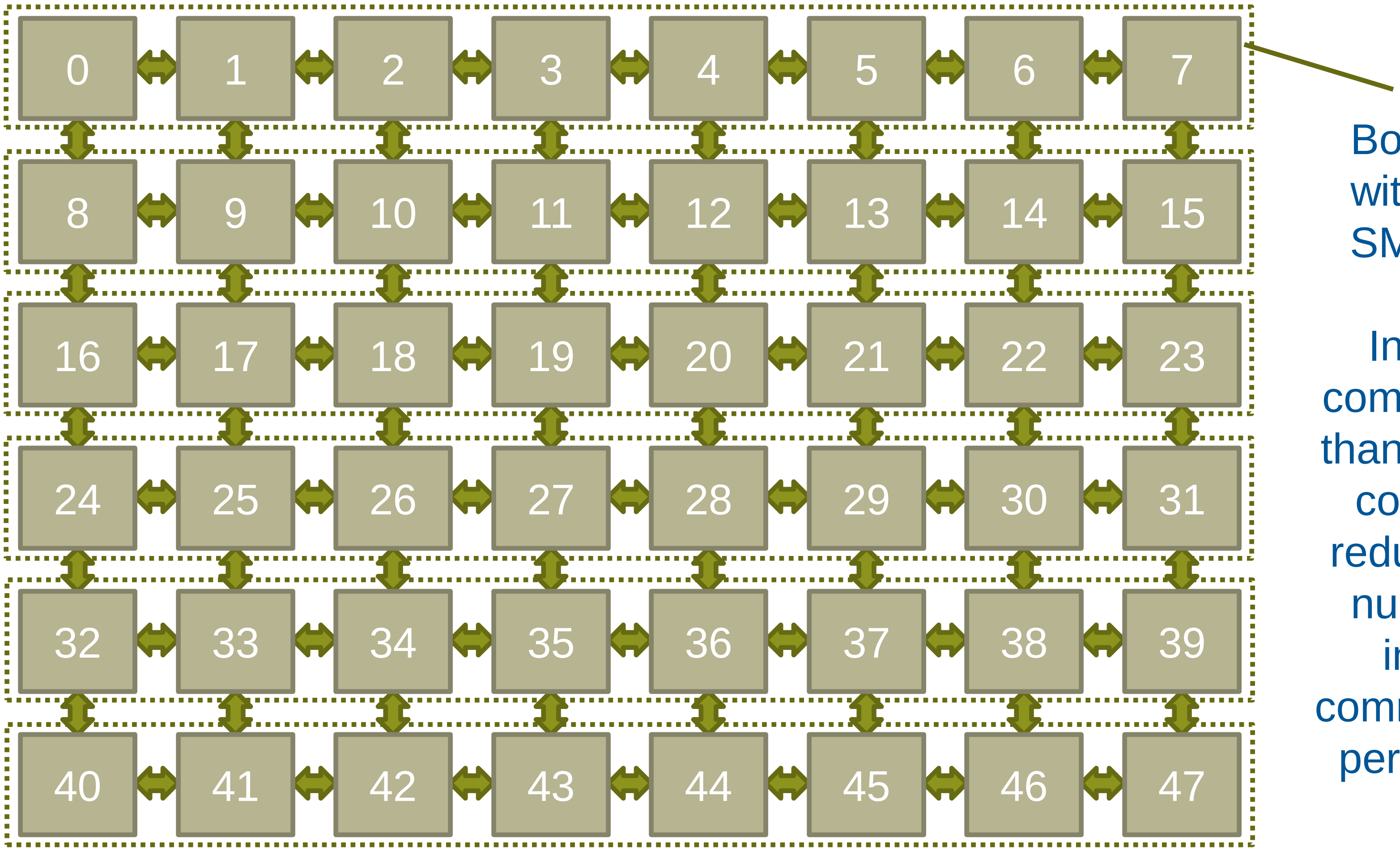
- **So easy to experiment with that the defaults at least should be tested with every application…**
- **When is this a priori useful?**
  - Point-to-point communication consumes a significant fraction of program time and a load imbalance detected
  - Also shown to help for collectives (alltoall) on subcommunicators
  - Spread out I/O servers across nodes

Optimising 2D Boundary Swap with Custom Rank Reorder



Each rank communicates with its N-S and E-W neighbours.

Default Rank Order: Suboptimal

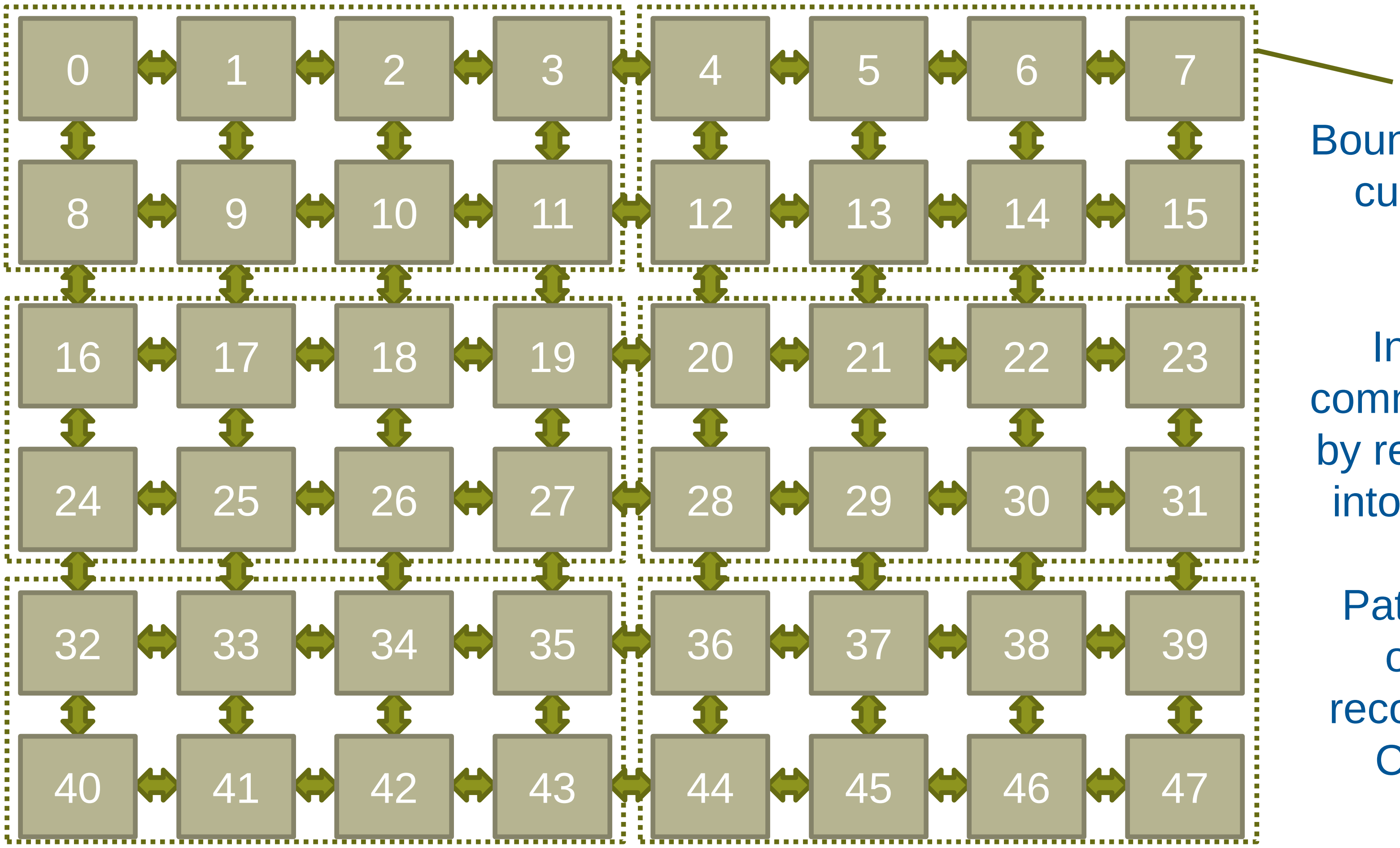


Node Boundaries with default SMP layout

Internode comms slower than Intranode comms, so reducing total number will improve communication performance

Default SMP layout creates 18 inter-node comm pairs per node

Improved Customised Order using sub-cells



Node Boundaries with customised layout

Internode comms reduced by reorganising into 4x2 cells.

Patterns can often be recognised by CrayPAT.

Customised ordering reduces to 12 inter-nodes.
Even more effective with 3D and fatter nodes.

Using `grid_order` to generate custom Rank Order files

The `grid_order` utility is used to generate a rank order list for use by an MPI application that uses communication between nearest neighbors in a grid. When executed with the desired arguments, `grid_order` generates rank order information in the appropriate format and writes it to `stdout`. This output can then be copied or written into a file named `MPICH_RANK_ORDER` and used with the

`MPICH_RANK_REORDER_METHOD=3`

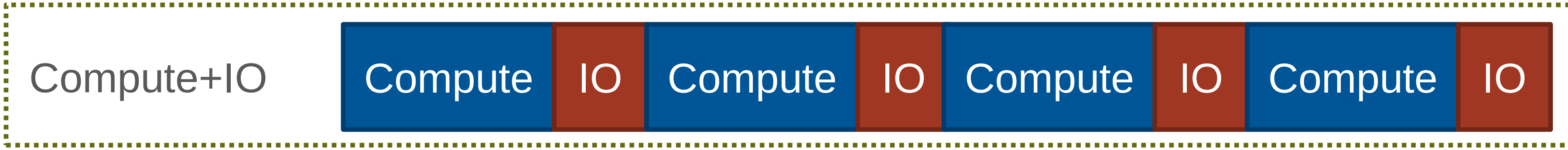
environment variable to override the default MPI rank placement scheme and specify a custom rank placement.

Combining Rank Reordering and MPMD mode

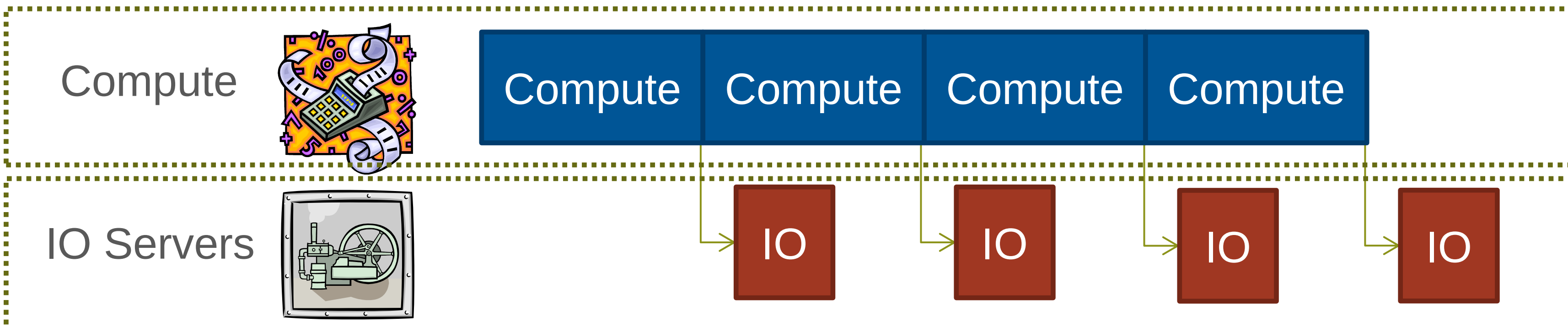
Inspired by a real world example

IO Servers – a quick recap

Originally codes treat compute and IO as serial tasks to be performed by all nodes

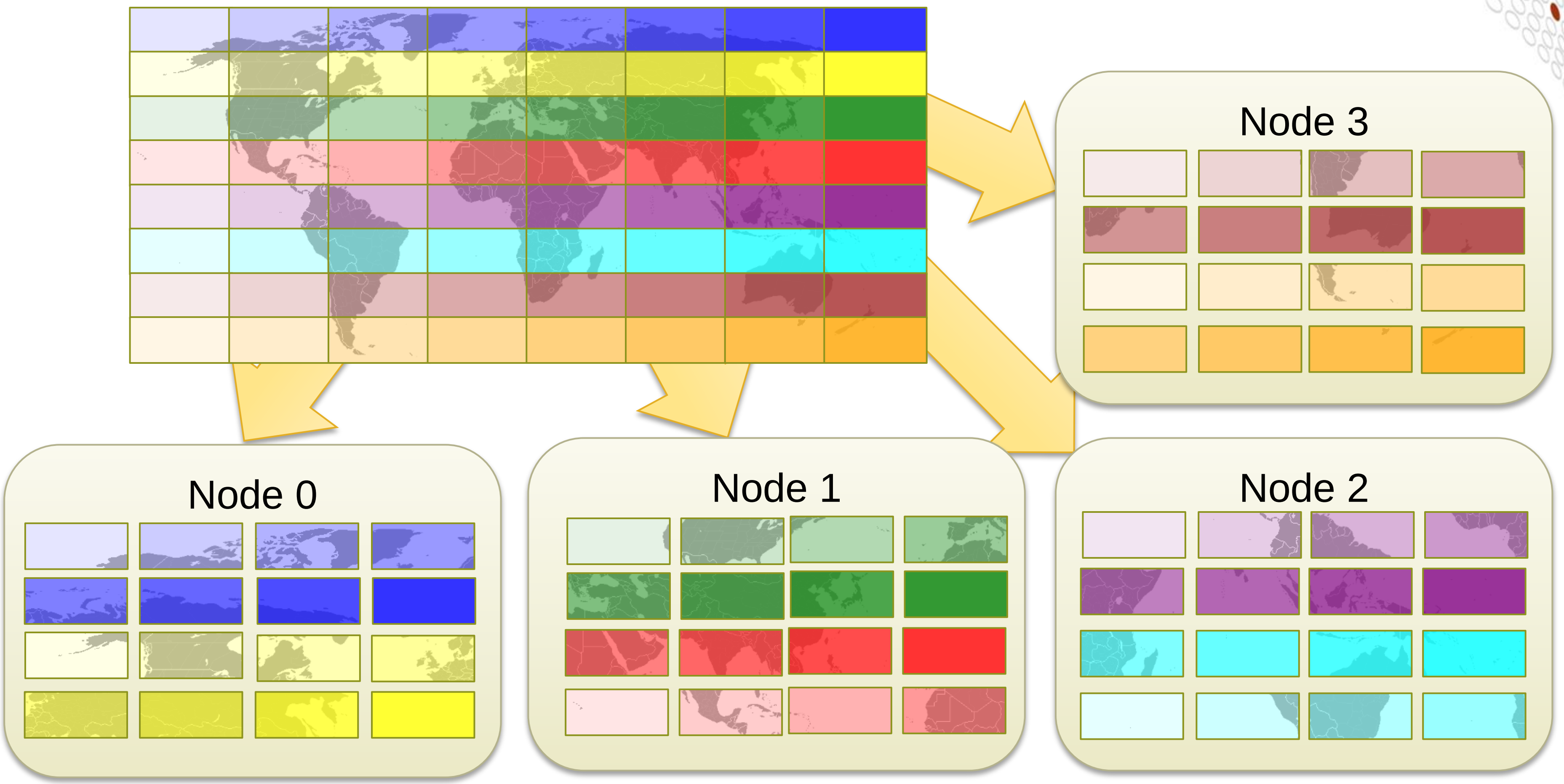


- IO costs have grown so codes (e.g. UM) have been extended to include IO Server ranks
- These ranks are dedicated to performing the IO operations asynchronously of compute.



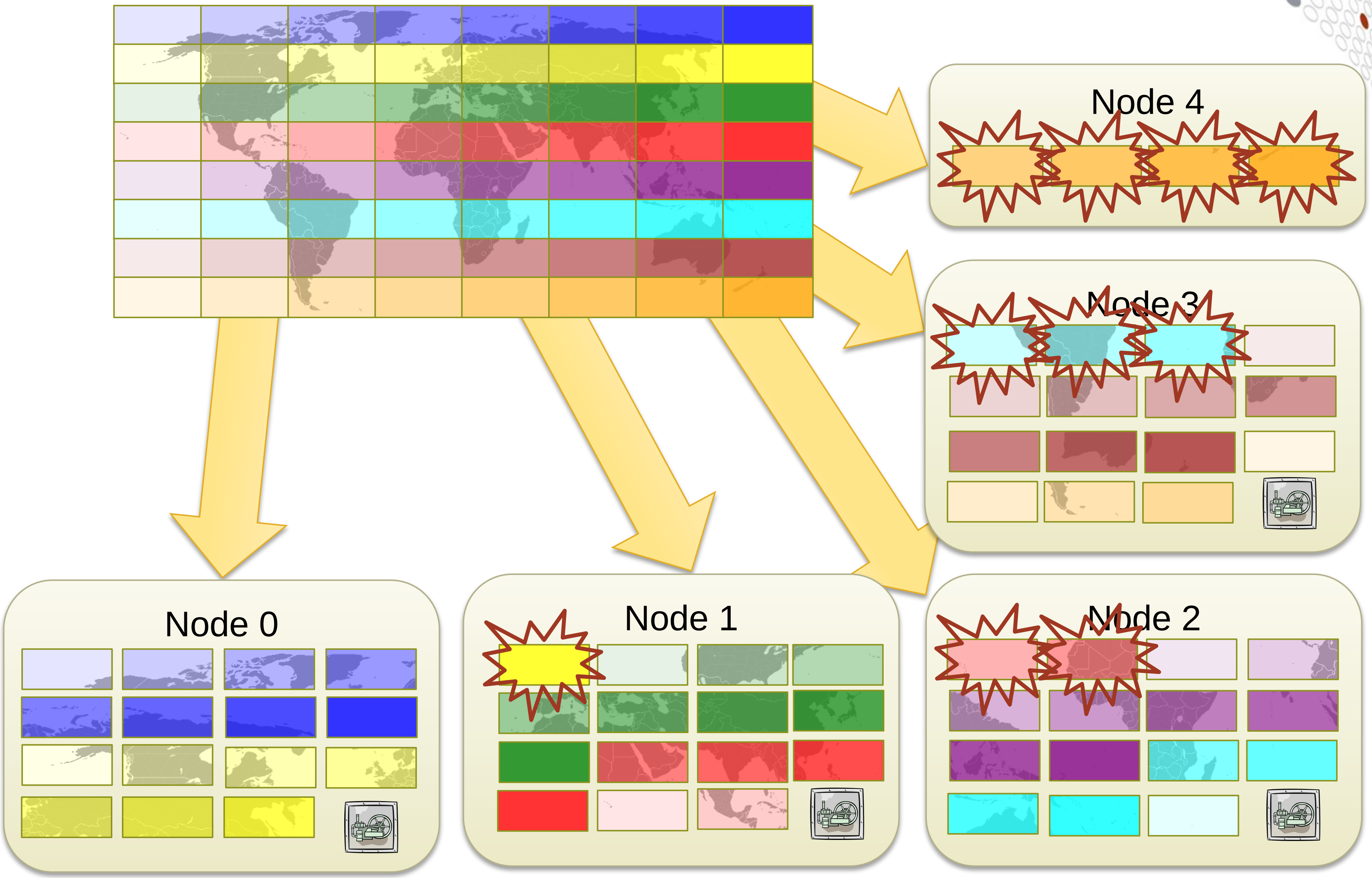
- Typically adding an additional 1% of nodes to act as IO servers can eliminate almost all IO from runtime.
- Requires data to be “double buffered”, so can increase overall memory overhead.
- Essentially a form of MPMD

A Standard MPI Domain Decomposition



- Domain decomposition distributed over the cores on each processor
- Deep East-West halos favour rows using intra-node comms (e.g. shared memory)
- Best performance achieved when processor E-W decomposition is a factor or domain E-W decomposition

Basic distribution of IO servers (1 per node)

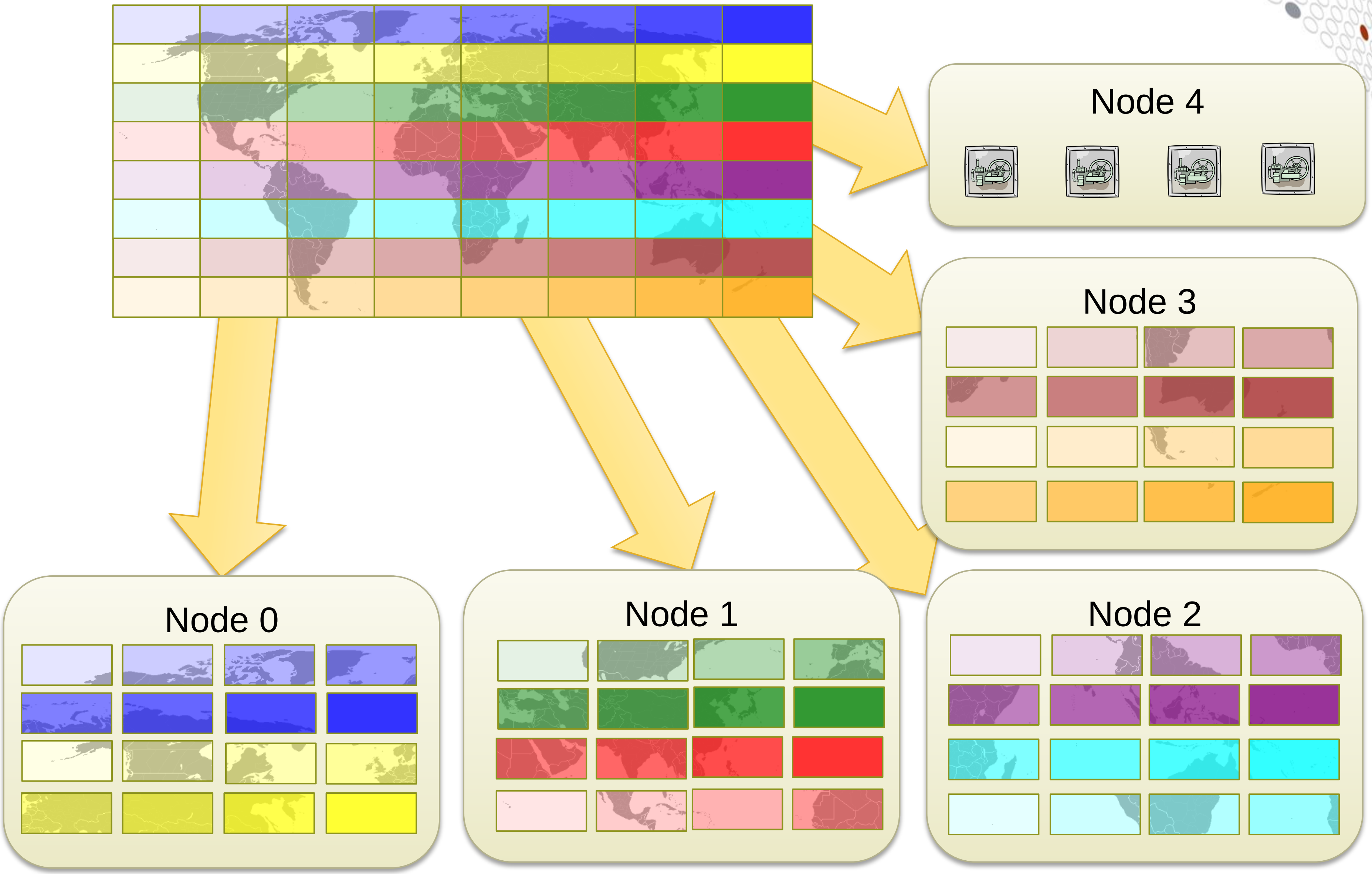


Basic distribution of IO servers (pt 2)

- **Advantages**
  - Easier implementation
  - Efficient when number nodes ≈ number of IO servers
  - Do not have to change the distribution of ranks across nodes
    - E.g, Keep 12 ranks per node, just add to the total number of ranks
    - Allows for much larger buffers on IO Server tasks
  - Distributes IO traffic across the network.

- **Disadvantages**
  - Disrupts the “nice” alignment between decomposition and nodes
  - IO Servers restricted to the same memory limits as compute ranks
    - IO Servers likely to require more memory, far less compute.

Rank Reordered Decomposition (IO Nodes)



Rank Reordered Decomposition (pt 2)

- **Advantages**
  - Keeps the “nice” alignment between proc decomposition and nodes
  - Can change the distribution of ranks across nodes
    - keep large numbers of ranks per node for compute nodes
    - use fewer ranks per node on IO nodes
  - Can be implemented at runtime with a custom `MPICH_RANK_ORDER` file
  - Most efficient when number compute nodes >> number of IO servers

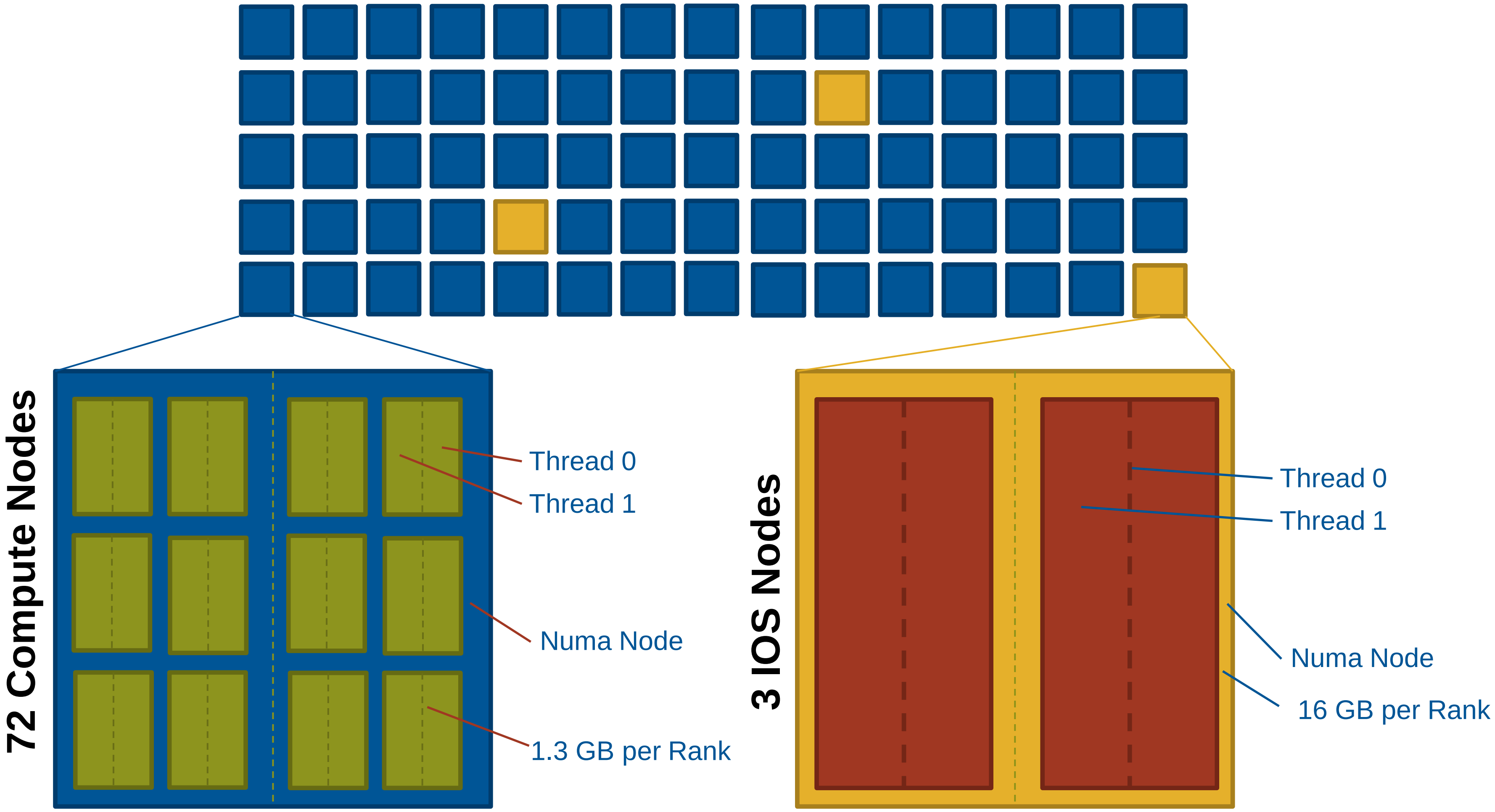
- **Disadvantages**
  - Concentrates IO traffic on a few nodes on the systems
    - However, network bandwith > IO Bandwidth
    - IO Servers should hide any IO delays anyway.

Providing more memory to IOS ranks

- **Most applications launch with a single set of aprun options**
  - Means every node (and usually every rank) is in a homogenous environment with the same number ranks per node.
  - Ideal for homogenous SPMD applications
- **aprun allows users to launch applications in MPMD mode**
  - allows users to launch applications with multiple sets options within a single MPI_COMM_WORLD communicator
  - Means ranks may have different runtime conditions, e.g. number of ranks per node, or strict memory containment
- **IOS ranks main requirements are large memory buffers, however compute ranks require much less.**
- **Using MPMD mode and rank reordering can create high memory IOS nodes and dense compute rank nodes.**
  - Also allows "nice" decomposition of compute nodes to continue

Example: 12x72x2 Compute Ranks + 6x2 IOS Ranks

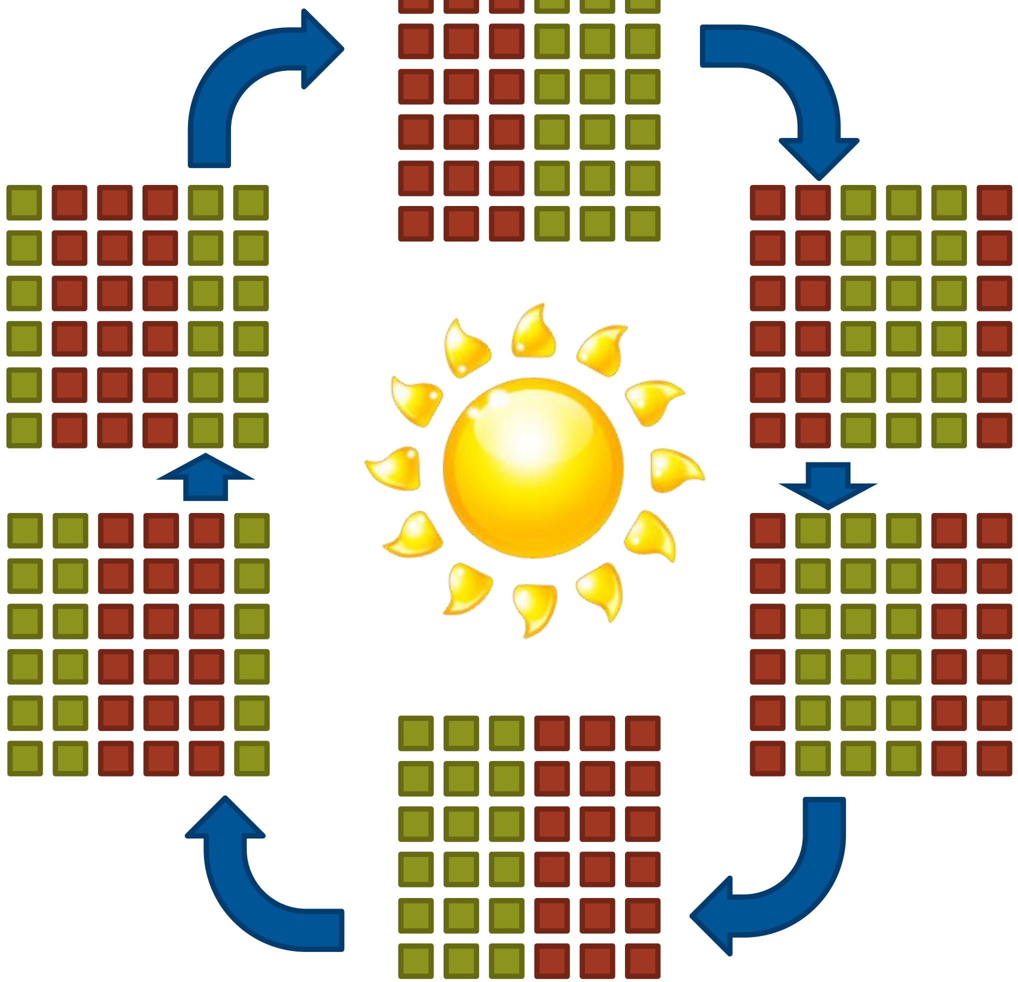
```
aprun -n 288 -N 12 -d 2 -j1 $EXE : -n 2 -N 2 -d 2 -S 1 -j1 $EXE :
      -n 288 -N 12 -d 2 -j1 $EXE : -n 2 -N 2 -d 2 -S 1 -j1 $EXE :
      -n 288 -N 12 -d 2 -j1 $EXE : -n 2 -N 2 -d 2 -S 1 -j1 $EXE
```



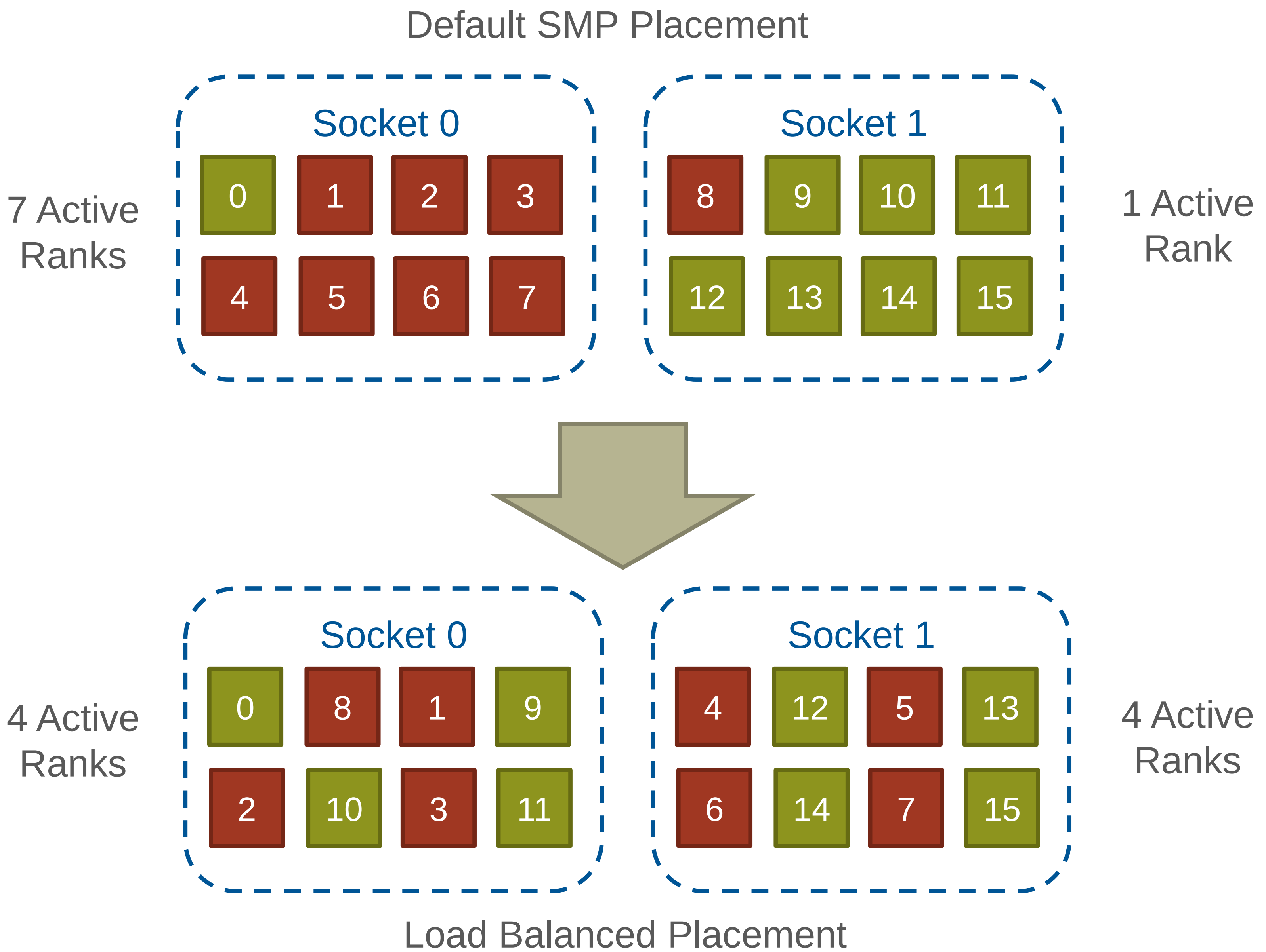
NUMA Node reordering

Reordering ranks within a node

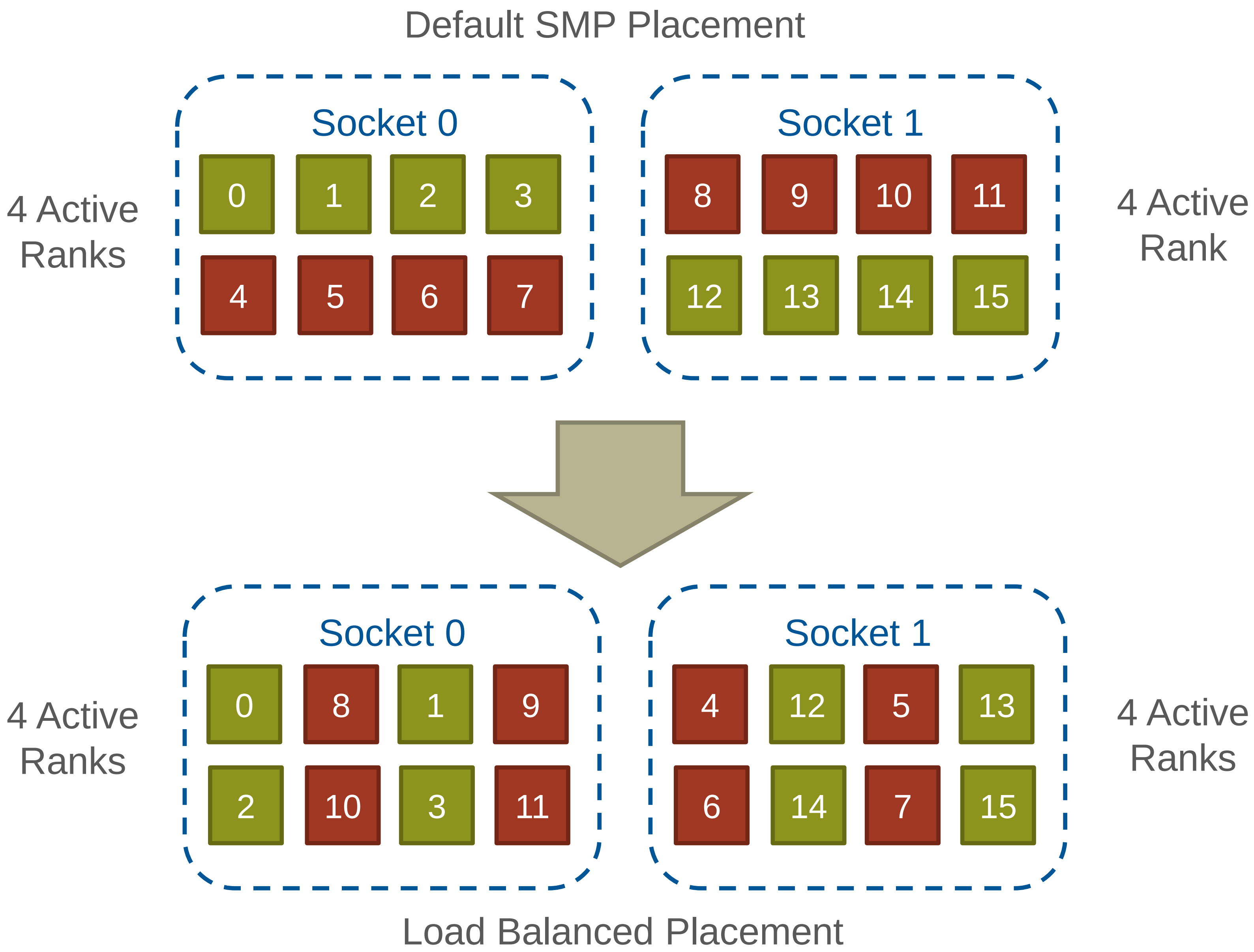
- Short-Wave radiation models are one of the more expensive sub-models
- However, only half the earth is lit at anyone time. This typically translates to only half the processors “active” during these phases
- With default SMP placement this means potential memory bandwidth imbalance across sockets



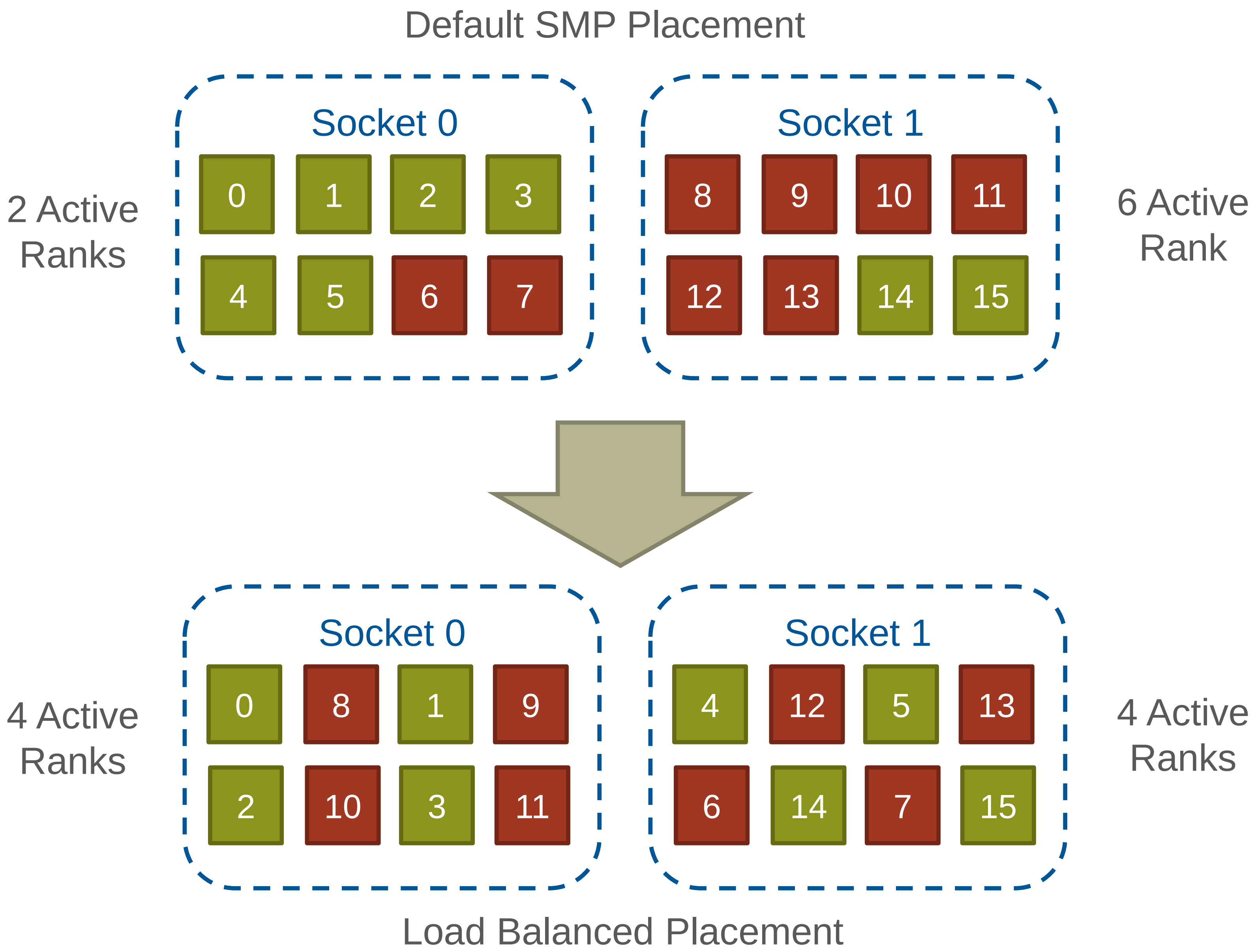
Socket Imbalance 1



Socket Imbalance 2



Socket Imbalance 3



Hybrid MPI + OpenMP?

- **OpenMP may help**
  - Able to spread workload with less overhead
  - Large amount of work to go from all-MPI to (better performing) hybrid - must accept challenge to hybridize large amount of code
- **When does it pay to add OpenMP to my MPI code?**
  - Add OpenMP when code is network bound
  - Adding OpenMP to memory bound codes may aggravate memory bandwidth issues, but you have more control when optimizing for cache
  - Look at collective time, excluding sync time: this goes up as network becomes a problem
  - Look at point-to-point wait times: if these go up, network may be a problem
  - If an all-to-all communication pattern becomes a bottleneck, hybridization often overcomes this
  - Hybridization can be used to avoid replicated data

OpenMP thread placement

- **When running a hybrid MPI+OpenMP application, the optimal number of threads/MPI task depends on the application and even input**
  - On the XC, one should try at least with 32x1, 16x2, perhaps also with 8x4, even 4x8 (MPI tasks x OpenMP threads per node)
- **The XE system is able to place OpenMP threads appropriately when the code is compiled with the Cray, PGI or GNU compiler**
  - Just do e.g. ”aprun -n 64 -d 32 -N 1 ./a.out” (for a 64x32=2048 core job)
  - You can use the aprun switch -S to force a certain number of MPI tasks per a numa node (=CPU) and -ss to have the threads to allocate memory only in the local numa node

Summary

- **Load imbalance is very often the very reason for non-scalability of an application**
- **It can be due to imbalanced computation or communication, with the usual suspects being**
  - Bad decomposition
  - All-to-one communication patterns
  - Single-writer I/O
- **Usually needs fixing at the source code level**
- **Some things for non-severe load imbalances can be done on the environment level: try to adjust the rank placement**
- **Hybrid MPI+OpenMP approach often useful for overcoming load balance problems**
  - Mind the thread placement when using hybrid codes!