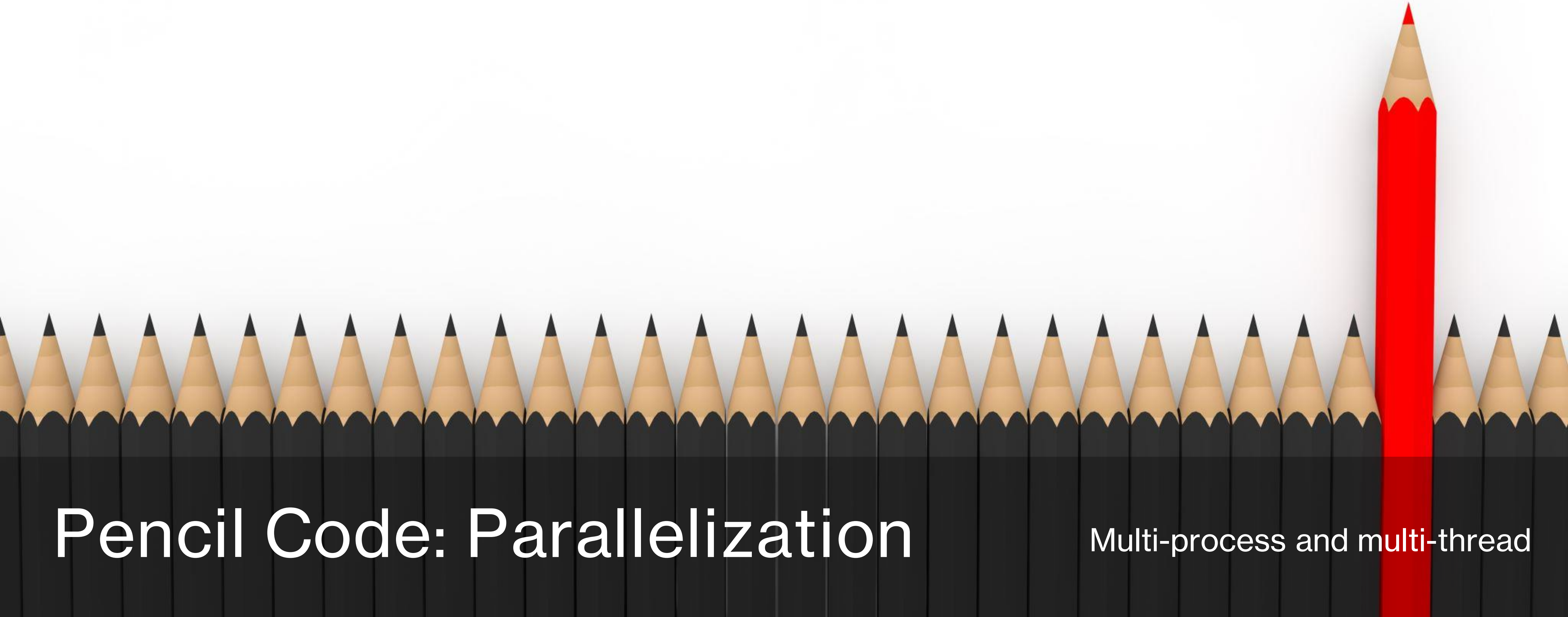
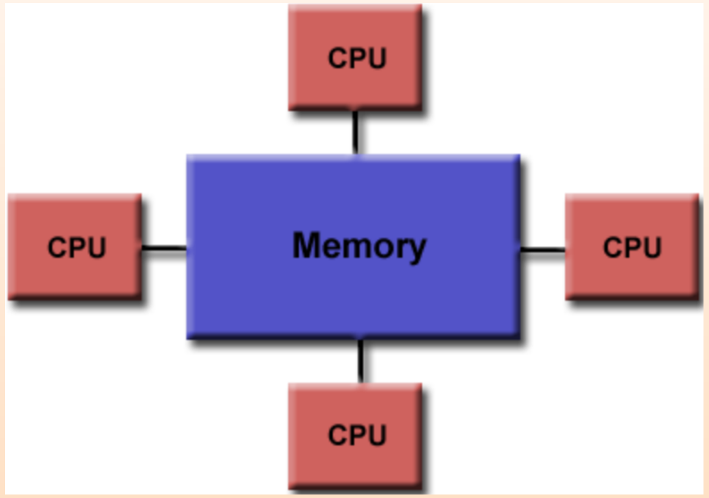


Pencil Code: Parallelization

Multi-process and multi-thread

Why does PC need multiprocess parallelization?



- medium-size setups, say 1024^3 grid points, 8 variables, double precision

  -> 128+ Gbyte main memory < on-node memory in big clusters

  -> shared memory, one-process approach

  (but time to solution with 64 ... 128 cores?)

- large-size setups, say 4096^3 grid points, ...

  -> 8+ TByte main memory >> on-node memory in big clusters

  -> distributed memory approach necessary -> multi-process implementation

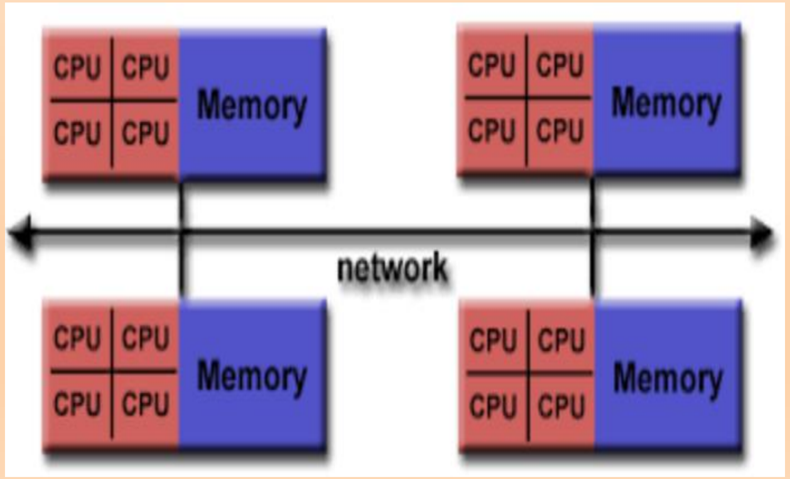
  naturally:

  SPMD paradigm (Single Program Multiple Data)

  = multiple copies of same program work on equally sized problem parts

  PDE: update of variables in a point needs information from local surroundings

  -> interprocess communication needed -> MPI



What is Message Passing Interface (MPI)?

- standardised library - different implementations (OpenMPI, MPICH)
- allows different UNIX processes to communicate
- -> in general: Multiple Program Multiple Data paradigm (MPMD),

  in particular SPMD: same executable runs in multiple instances
- messages = tagged data parcels exchanged within communicators

https://www-lb.open-mpi.org/software/ompi/v5.0/

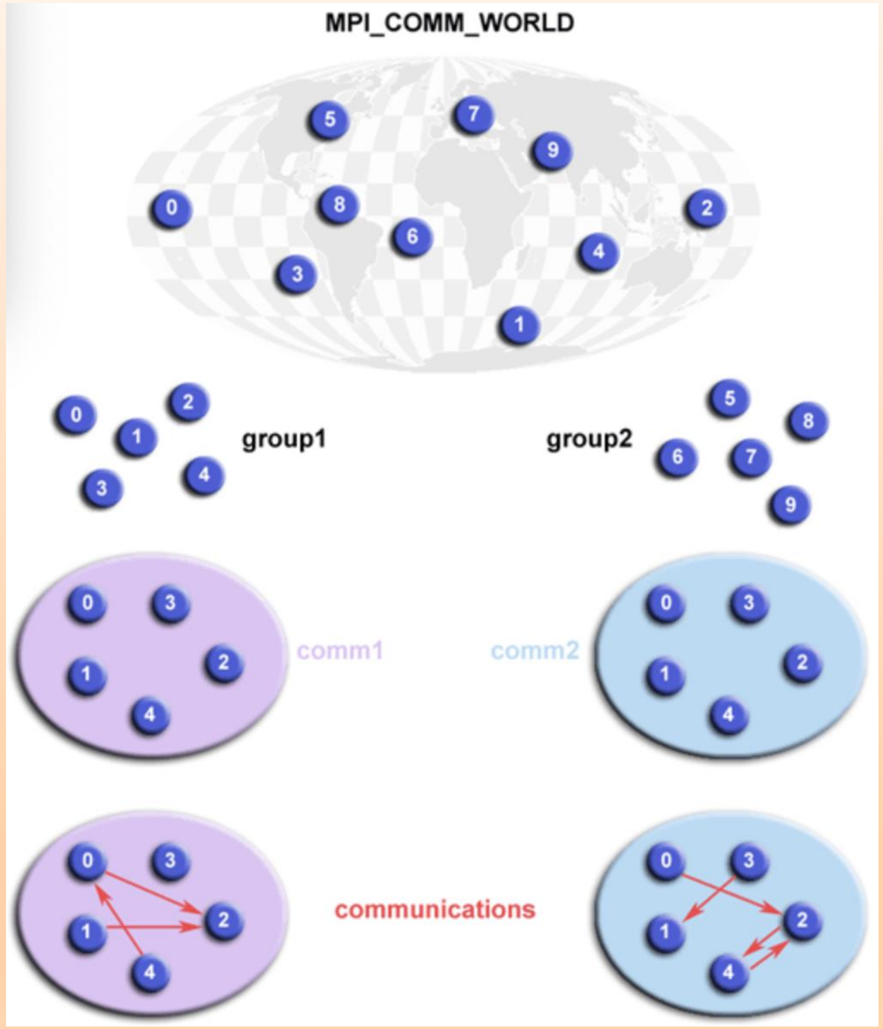
https://www.mpich.org/documentation/guides/

What is a communicator?

- group of processes with context (tag, topology, attributes)
- exclusive: messages sent within one comm. cannot be received in another comm. (exception: intercommunicators)
- default: MPI_COMM_WORLD

  = all processes launched together by

  mpirun, mpiexec, srun etc.



Point-to-point vs. Collective communications

- Point-to-point (P2P): one sending proc -- one receiving proc

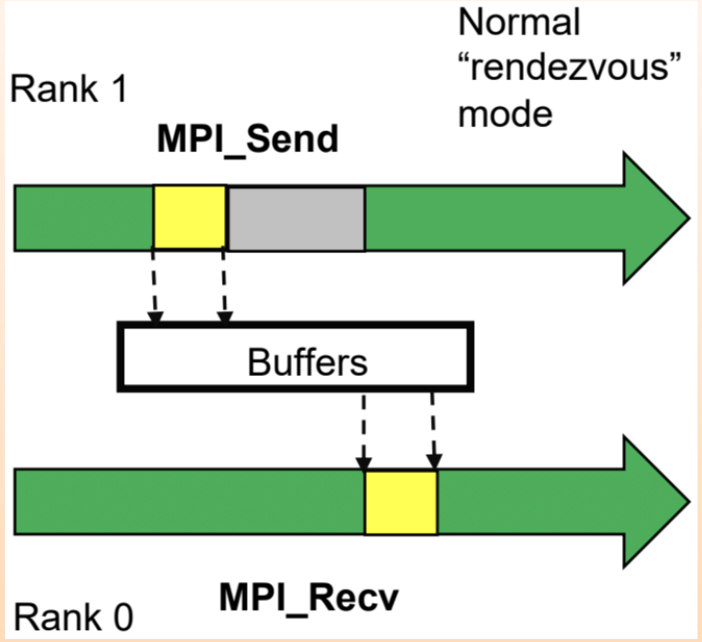
  (but simultaneous send and receive with one call MPI_SENDRECV)

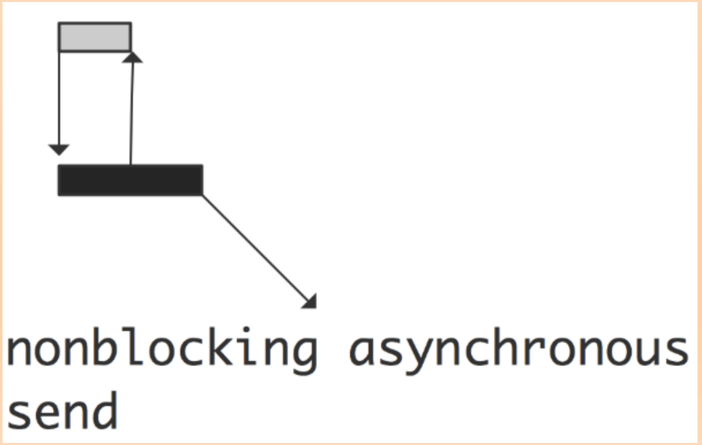
- collective: all procs of a comm. must execute the call,

  typically reductions like sum or max/min

- blocking vs. non-blocking communication calls:
  - blocking: return only if a "success criterion" is fulfilled (MPI_[B|R|S|]SEND)
  - non-blocking: return immediately (MPI_I[B|R|S|]SEND)

    -> key for concurrency of communication and computation (or other)





Implementation in Pencil Code

Example: 8x6x6 grid

- processes arranged as a Cartesian "process grid",

  defined by the user through `nproc[xyz]` in cparam.local

  -> each process has "coordinates" `iproc[xyz]` in this grid

  (Cartesian topology could be enforced by MPI tools, but is not at the moment)

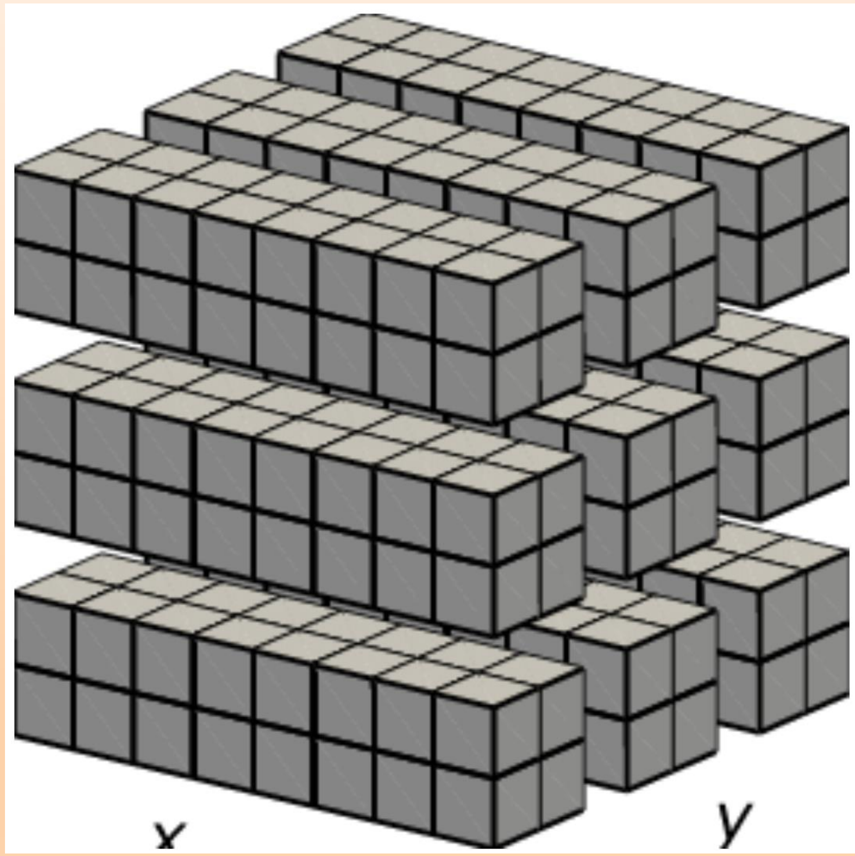
- computational grid is divided in `ncpus` equally sized

  cuboid subdomains according to `nproc[xyz]`

  subdomain size:

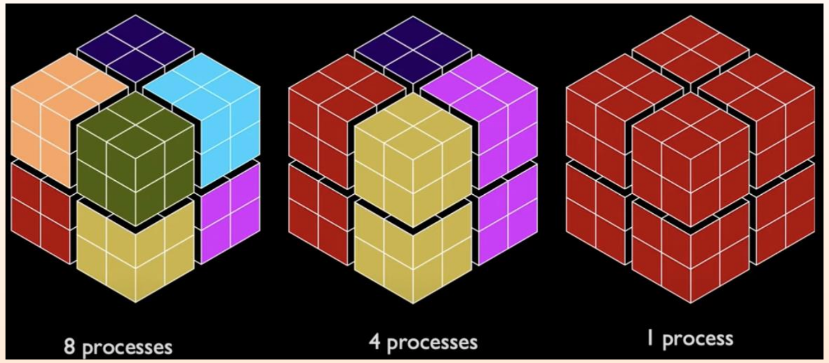
  `(nxgrid/nprocx, nygrid/nprocy, nzgrid/nprocz)`

- each process holds a subdomain



`nprocx=1, nprocy=3, nprocz=3`

Implementation in Pencil Code



- task:

  mapping process numbers (“ranks”, in PC: `iproc`) to subdomains

  processes linearly numbered: 0, 1, 2, ... total number of procs-1

  how to map `rank -> (iprocx,iprocy,iprocz)` ?

- optimal: union of the subdomains of all processes on

  a compute node as close to a cube as possible

  -> maximized on-node communications

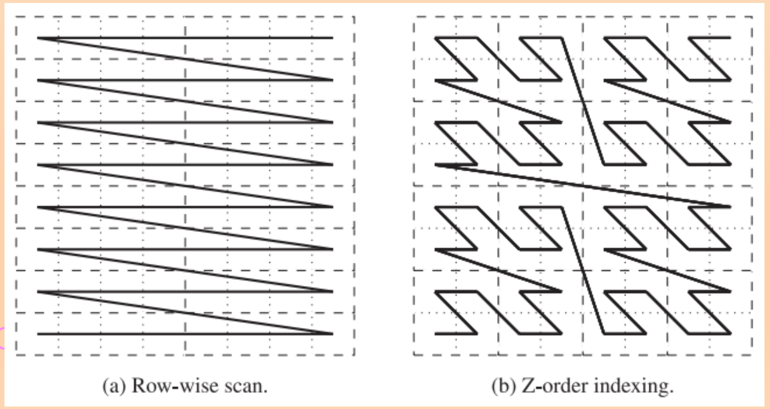
- at the moment naive linear map with

  `iprocx` running fastest, `iprocz` slowest:

  ```
  iproc = iprocx + iprocy*nprocx + iprocz*nprocx*nproc
  ```

- better: Morton-curve numbering

4 processes per node



1-2 partners on node 3

Major communication task: from stencils

- stencil structure of finite difference formulae:
  - for most differential operators

    3-dimensional von-Neumann stencil with radius r=3

    (default, other options: 1, 2, 4, 5)

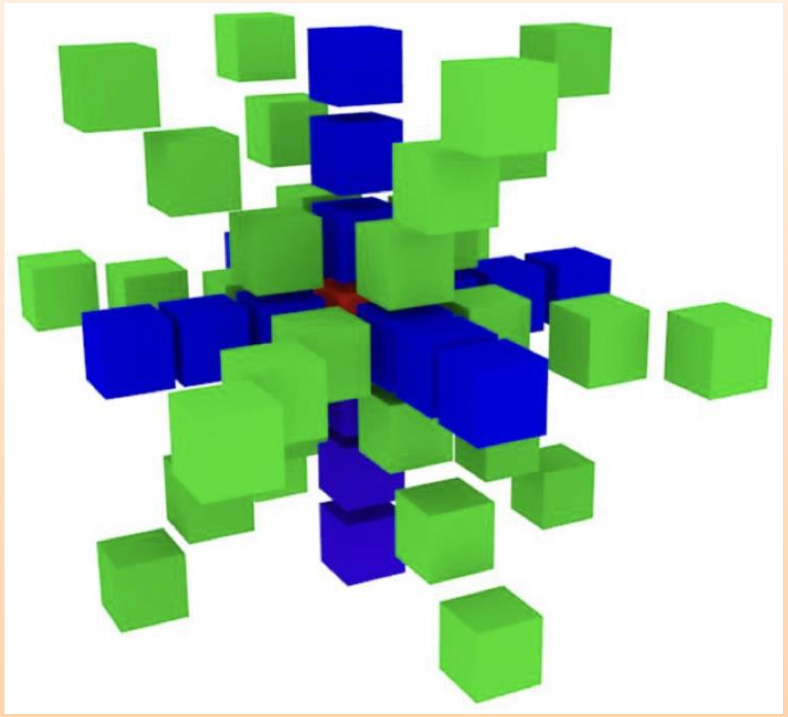
  for mixed derivatives $\frac{\partial^2}{\partial_{x_i}\partial_{x_j}}, i \neq j$:

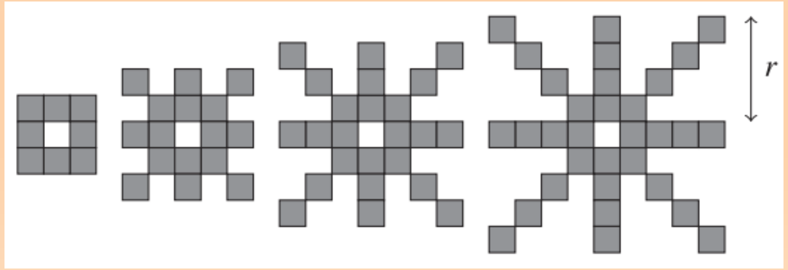
  2-dimensional Moore stencils in xy, xz and yz planes,

  but simplified to

  2-dimensional von-Neumann stencils w. 45° rotation

default 55-point 3D stencil in PC





2D PC-stencil r =1,2,3,4

Major communication task: halo update

- stencil operations

  -> per-process grid subdomains need a halo ("ghost zones")

  outside subdomain = inside subdomains of other processes

  ("inner halo")

- dictates "halo update" or "ghost zones update" through

  interprocess communication

  for each variable updated by stencil operations

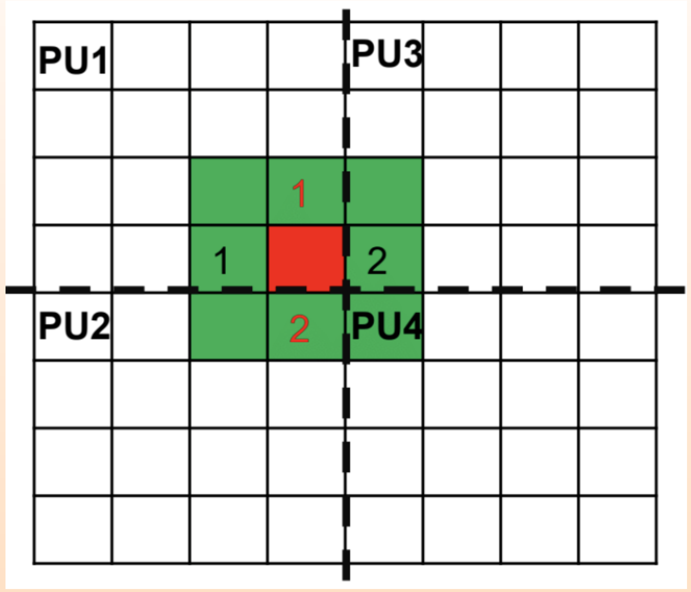
  - all dependent variables of the PDEs

    all auxiliary variables subject to stencil operation (smoothing, maximum ...)

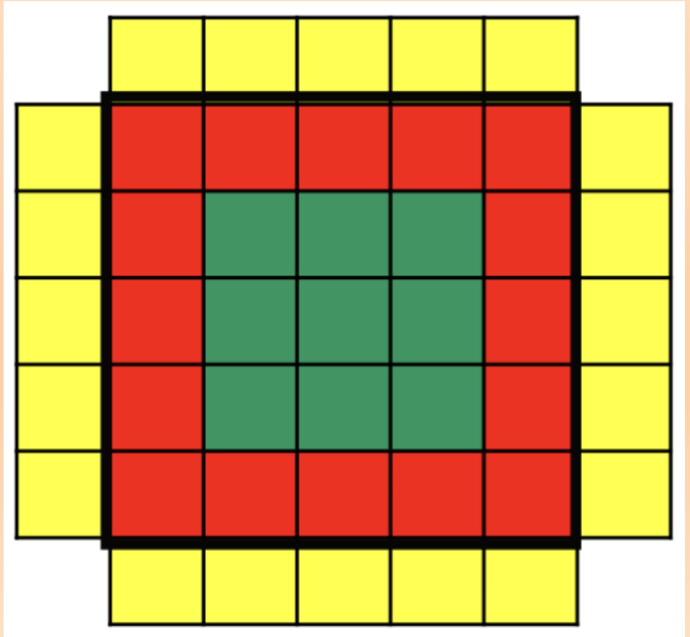
  - -> distinction between communicated and non-communicated auxiliaries

```
MAUX CONTRIBUTION     --      COMMUNICATED AUXILIARY
```

Moore r=1 stencil



von Neumann r=1 stencil



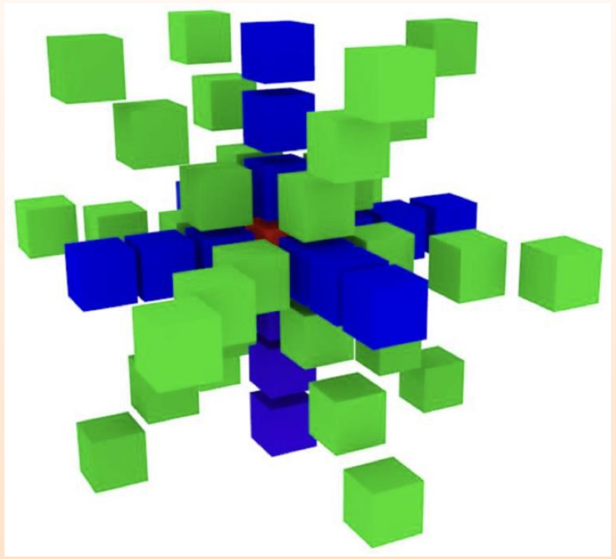
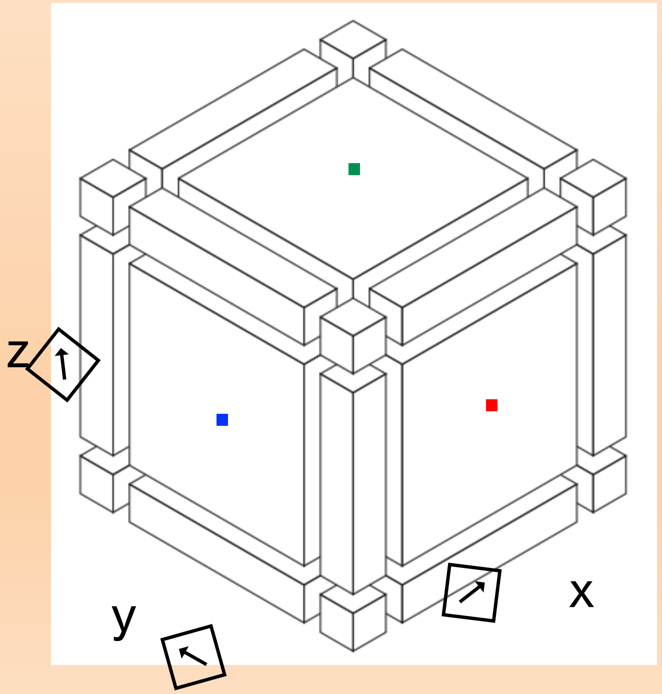
Halo update

- correspondingly: communication with
  - cross-face
  - cross-edge

  process neighbours, not with cross-vertex neighbours
- -> 6 cross-face process neighbours, communication conveys

left and right yz-plates	in positive and negative x-direction
front and back xz-plates	in positive and negative y-direction
top and bottom xy-plates	in positive and negative z-direction

- -> 12 cross-edge process neighbours, theoretically

Halo update

- but only 4 cross-edge communications = those in yz planes
  - 4 x-beams in

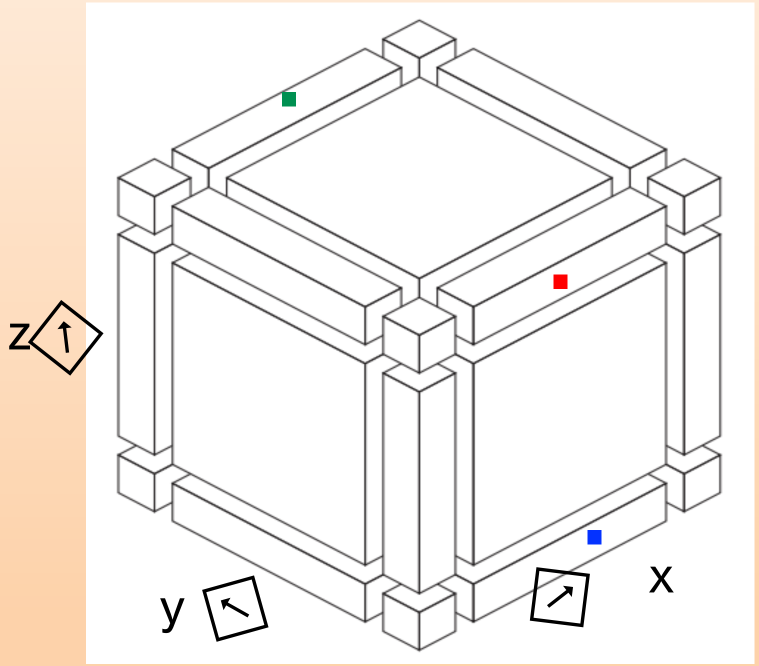
lower-lower	(-y, -z)
lower-upper	(+y, -z)
upper-upper	(+y, +z)
upper-lower	(-y, +z)

directions (yz plane)

- remaining beams (y and z):
  - cross-edge neighbour = cross-face neighbour of cross-face neighbour

    -> finish x-communication first,
  - let xy and xz plates also comprise ghost zone in y and z
- no communication across faces/edges of the global domain,

  unless they belong to a periodic direction
- instead: application of physical boundary conditions

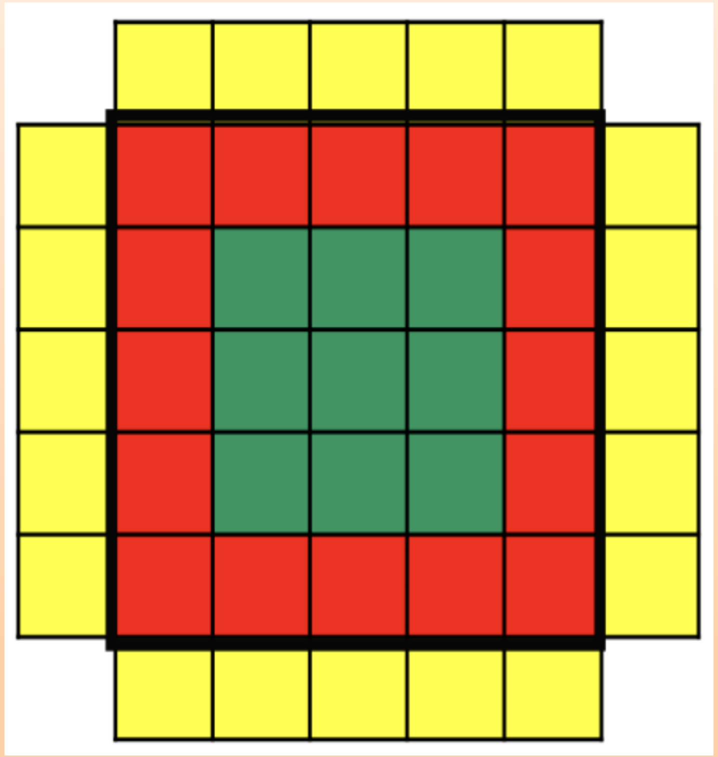


SPMD program flow with halo communication

- in each integration substep
  - fill communication buffers with data from "inner halo"
  - initialise point-to-point non-blocking communication

    = `call MPI_ISEND` halo-section-specific tag for all neighbours
  - call corresponding `MPI_IRECV`
  - update inner part of subdomain (independent of halo)
  - finalize point-to-point non-blocking communication

    = `call MPI_WAIT for each MPI_ISEND/MPI_IRECV`
  - after completion, update outer part of subdomain = inner halo
  - synchronise processes (`MPI_BARRIER`) and repeat



Further MPI-parallelized tasks

- reductions
  - averages/integrals over volumes, planes or lines (=0D to 2D averages)
    and global extrema for diagnostics: low cadence -> cheap
  - 1D or 2D averages for special integro-differential equations
    (↗ test methods): every substep -> expensive
  - 0D averages for conservation-preserving measures:
    remove mass/flow/momentum drift: adjustable cadence -> medium exp.
  - for separately studying large- and small-scale instabilities: ~
  - for special SGS models acting only on fluctuations: every substep -> expensive
  - implemented by blocking collective MPI calls
- spectra as diagnostics or for Poisson solving/BCs: parallelized FFT cheap -> expensive
- parallel write into monolithic snapshot file

MPI in Pencil Code: `mpicomm.f90`

- encapsulates all calls into MPI library

  -> for non-parallelised runs

     `nompicomm.f90` is included in the build instead
  - implements all preparatory work:
  - initialising/finalizing MPI
  - creating communicators
  - establishing neighbours
  - establishing "boundary process" flags
    `l[first|last]_proc_[xyz]`
  - allocating communication buffers and tags

```
public :: mpirecv_logical,
mpirecv_real,
                  mpirecv_int,
mpirecv_char   !,
                  mpirecv_cmplx
                  mpisend_logical,
mpisend_real,
                  mpiscan_int,
mpisend_int,
                  mpisend_char,
                  mpisendrecv_real,
                  mpisendrecv_int,

mpireduce_sum_int,
                  mpireduce_sum

mpireduce_sum_double
                  mpireduce_max,

mpireduce_max_int,
                  mpireduce_min
                  mpiallreduce_max,
                  mpiallreduce_min
                  mpiallreduce_sum,
```

Pencil Code communicators

- `MPI_COMM_WORLD` (default MPI-provided)

  in PC only used at initialisation and for MPMD
- `MPI_COMM_PENCIL`

  by default duplicate of `MPI_COMM_WORLD`, but can differ in MPMD mode
- `MPI_COMM_GRID`

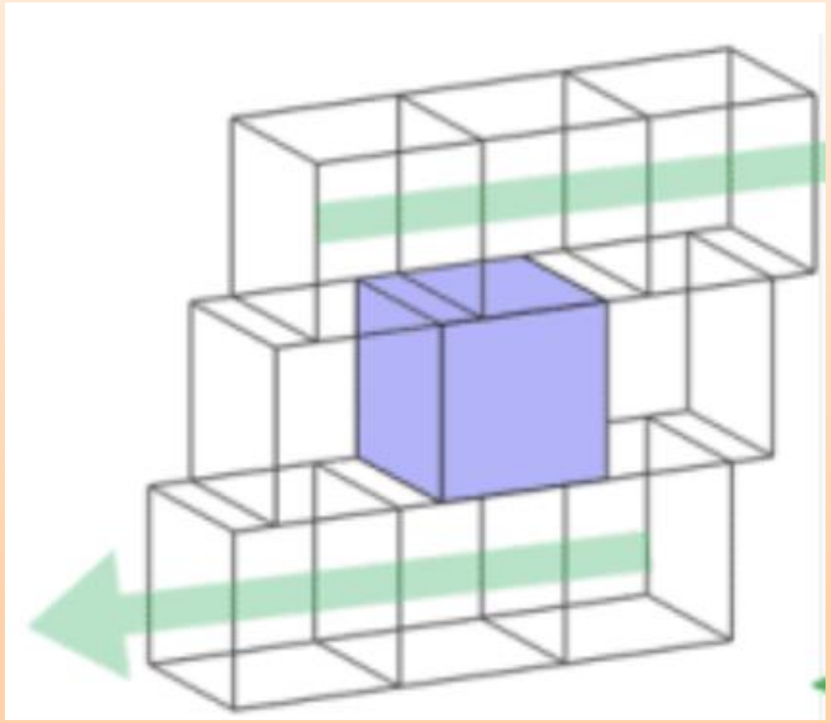
  by default duplicate of `MPI_COMM_PENCIL`,

  but split in two if PC holds two congruent grids (Yin-Yang grid)
- `MPI_[XY|XZ|YZ]PLANE` as many as there are XY|XZ|YZ planes in processor grid
- `MPI_[XYZ]BEAM` as many as there are X|Y|Z beams in the processor grid

Non-standard communication patterns: shear

- for “sliding periodic” boundary conditions (shearing-box):

  up to five neighbours at x-boundaries along y-direction

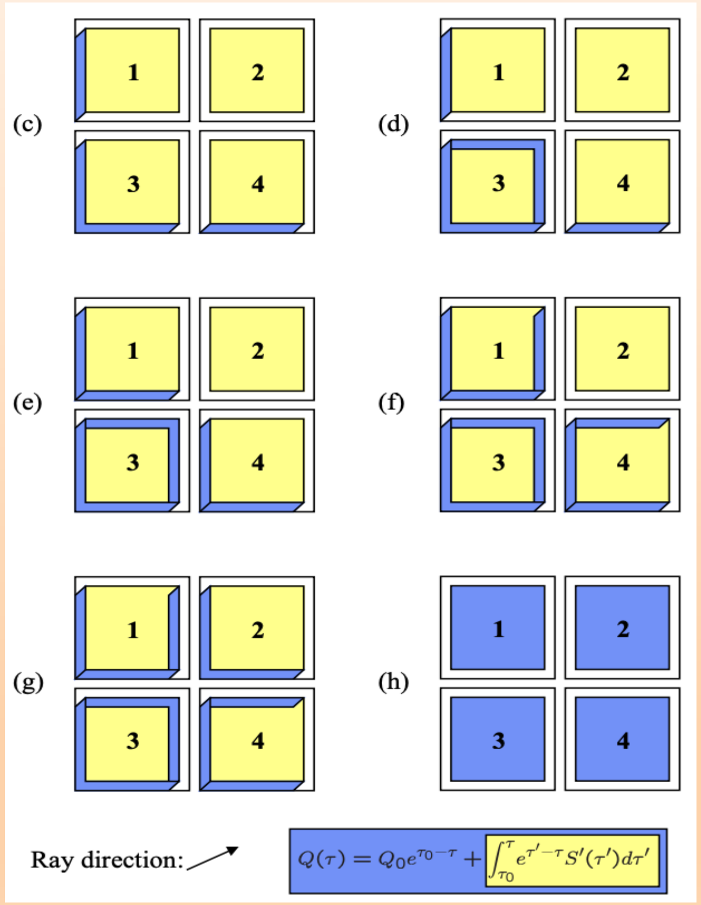


Non-standard communication patterns: radiative transport

- rays traverse whole domain, possibly several times for periodic BCs
- contributions to energy equation: line integrals along ray

$$I(\tau) = I(\tau_0)e^{-(\tau-\tau_0)} + \int_{\tau_0}^{\tau} e^{-(\tau-\tau')} S(\tau')\, d\tau' \,.$$

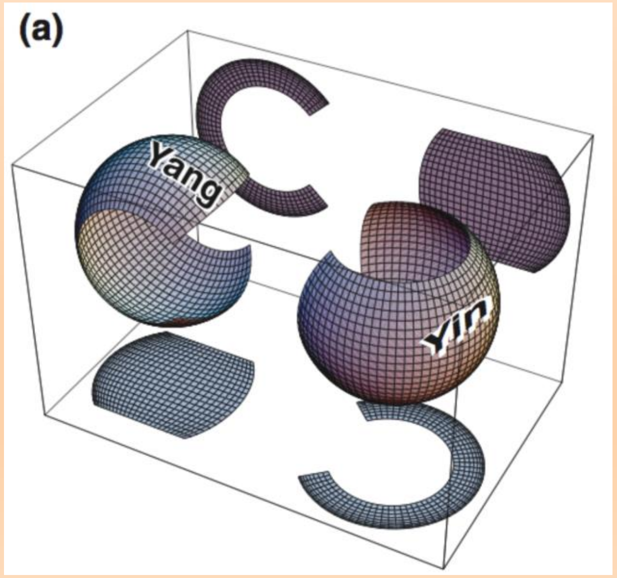
-> reductions needed

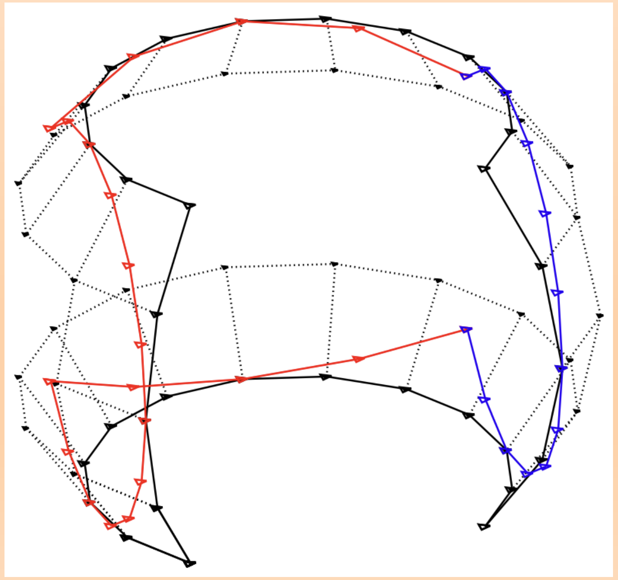

(c)
(d)
(e)
(f)
(g)
(h)
1
2
3
4
Ray direction:
$Q(\tau) = Q_0 e^{\tau_0-\tau} + \int_{\tau_0}^{\tau} e^{\tau'-\tau} S'(\tau') d\tau'$

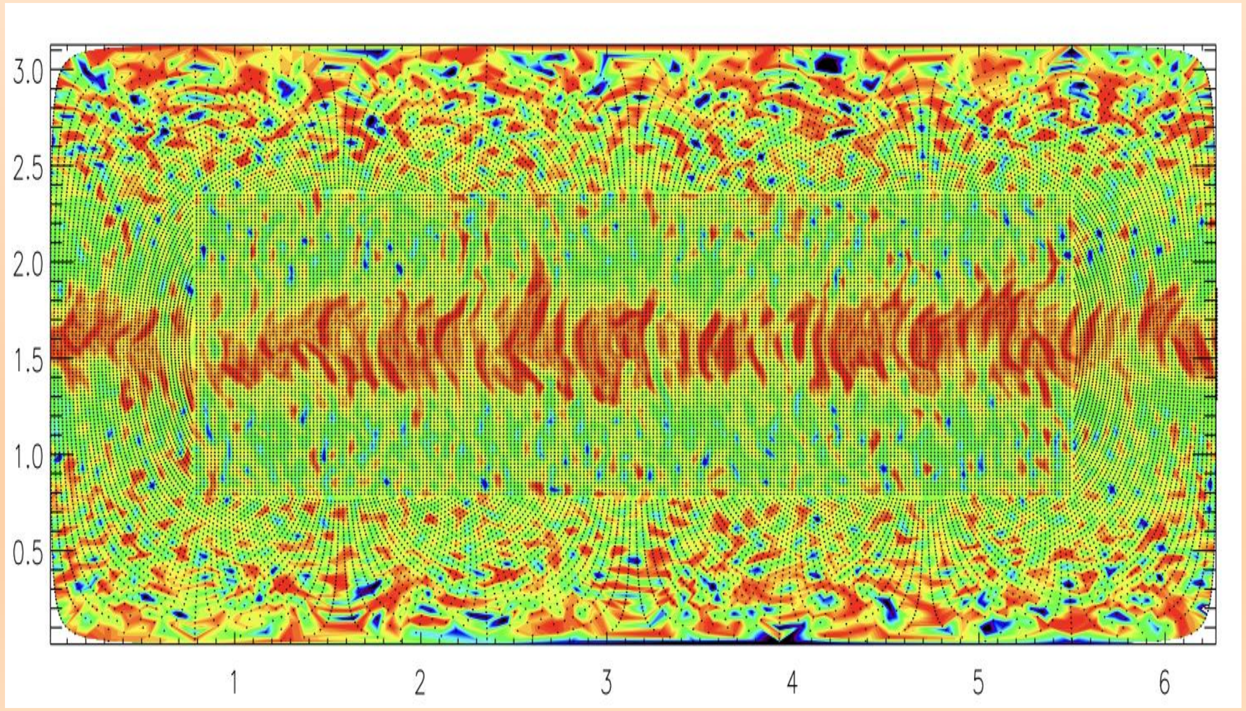
Non-standard communication patterns: Yin-Yang grid

- two congruent grids covering full sphere (communicator MPI_COMM_GRID)
- inside each grid: standard communication pattern

  at edges: defined by grid overlap
- yet experimental







Non-standard communication patterns: cross-pole

- 3D setups in spherical coordinates (r, ϑ, φ):

  axis singularity because of metric coefficient $1\,/r\sin\vartheta$

- can be avoided by $d\vartheta/2$ distance from pole

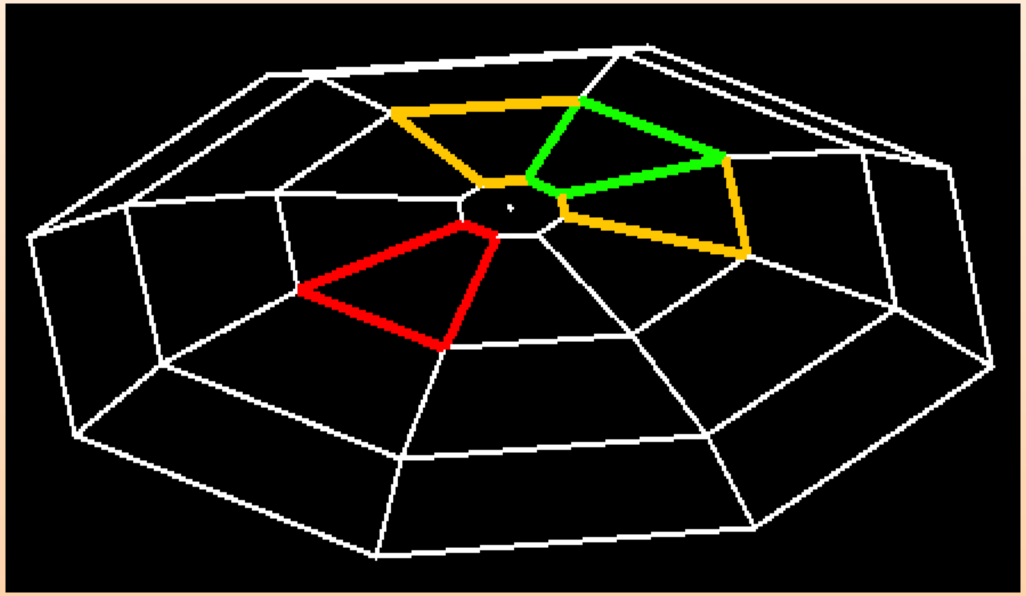
  -> a ϑ boundary?

  better not! artificial!

- alternative: consider “periodicity” across the pole

  -> all “y = cross-pole neighbors”

  have same `ipy` proc coordinate, but different `ipz`



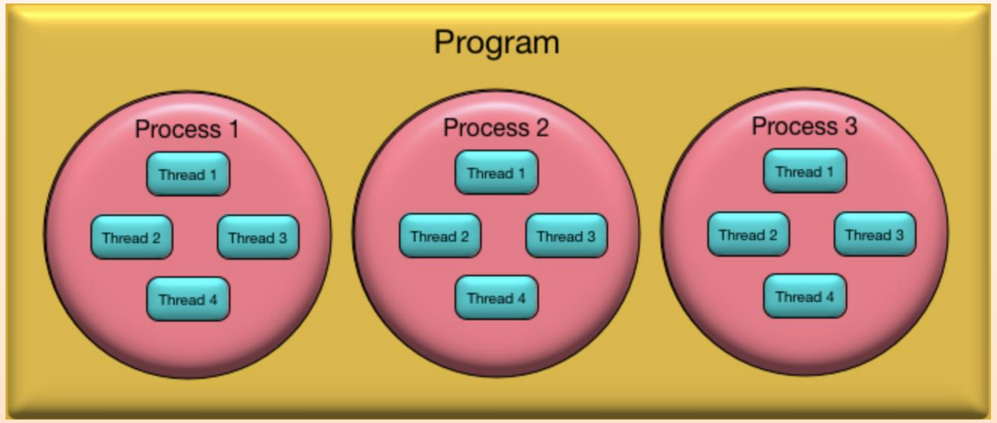
- tb combined with grid coarsening near poles – yet experimental

Non-standard communication patterns: MPMD

- interface to communicate with “foreign” grid-based code

  with independent processor layout
  - gathers information about foreign grid and layout
  - establishes communication pattern with foreign code
  - receives and interpolates data in full domain
  - yet experimental

Multithreading



- enables direct utilization of shared memory on node

  (MPI silently takes advantage of it, too)
- instead of multiprocess

  -> parallel threads of a single process on a group of CPU cores
- meaningful in connection with GPU acceleration:
  - for example: 8 GPUs, 64 CPU cores per node

    -> 8 MPI processes with 8 threads each, using all 64 cores
  - accelerates diagnostics calculation on CPU
- allows concurrence of diagnostics calculation and output with integration on GPUs

Multithreading

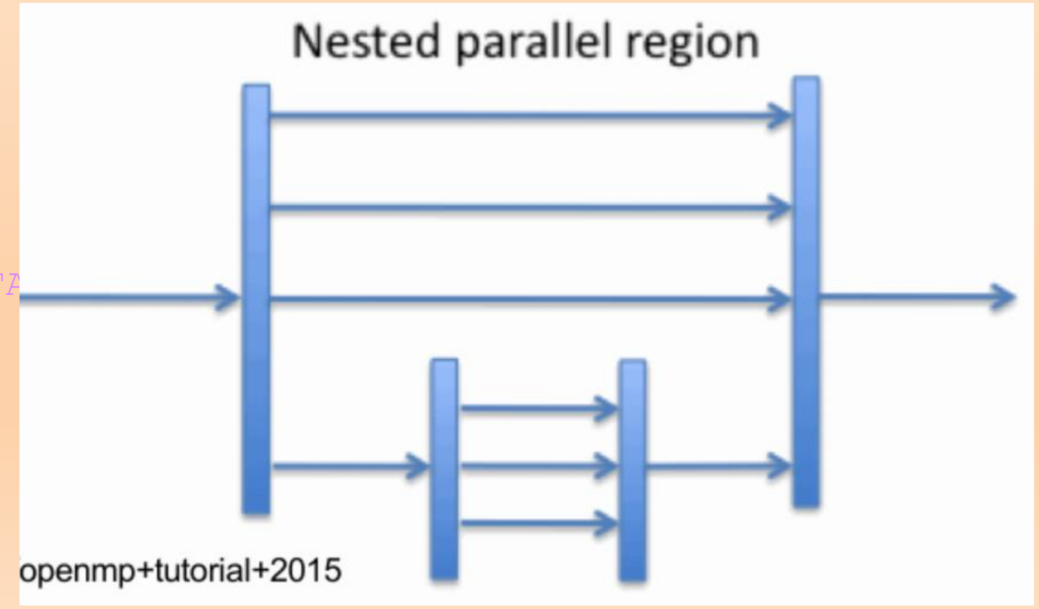
- implementation: nested parallelism
  - first parallel region with 2 threads: master <-> “diagmaster”

    master administers program flow, especially kernel launch on GPU

    diagmaster ~ diagnostic calcs and output of diagnostics and snapshots

    spawns new parallel region with all remaining available threads
  - encoded in Fortran by OpenMP directives (`!$omp` ...) and guards (`!$` ...), activated by proper compilation flags
  - runtime specifications, e.g. in SLURM:

```
#SBATCH --cpus-per-task=8
            …
           export OMP_NUM_THREADS=${SLURM_CPUS_PER_TA
           export OMP_MAX_ACTIVE_LEVELS=2
           export OMP_PROC_BIND=close,spread
           export OMP_WAIT_POLICY=PASSIVE
           pc_run
```


Nested parallel region
openmp+tutorial+2015