Parallel computing
An introduction

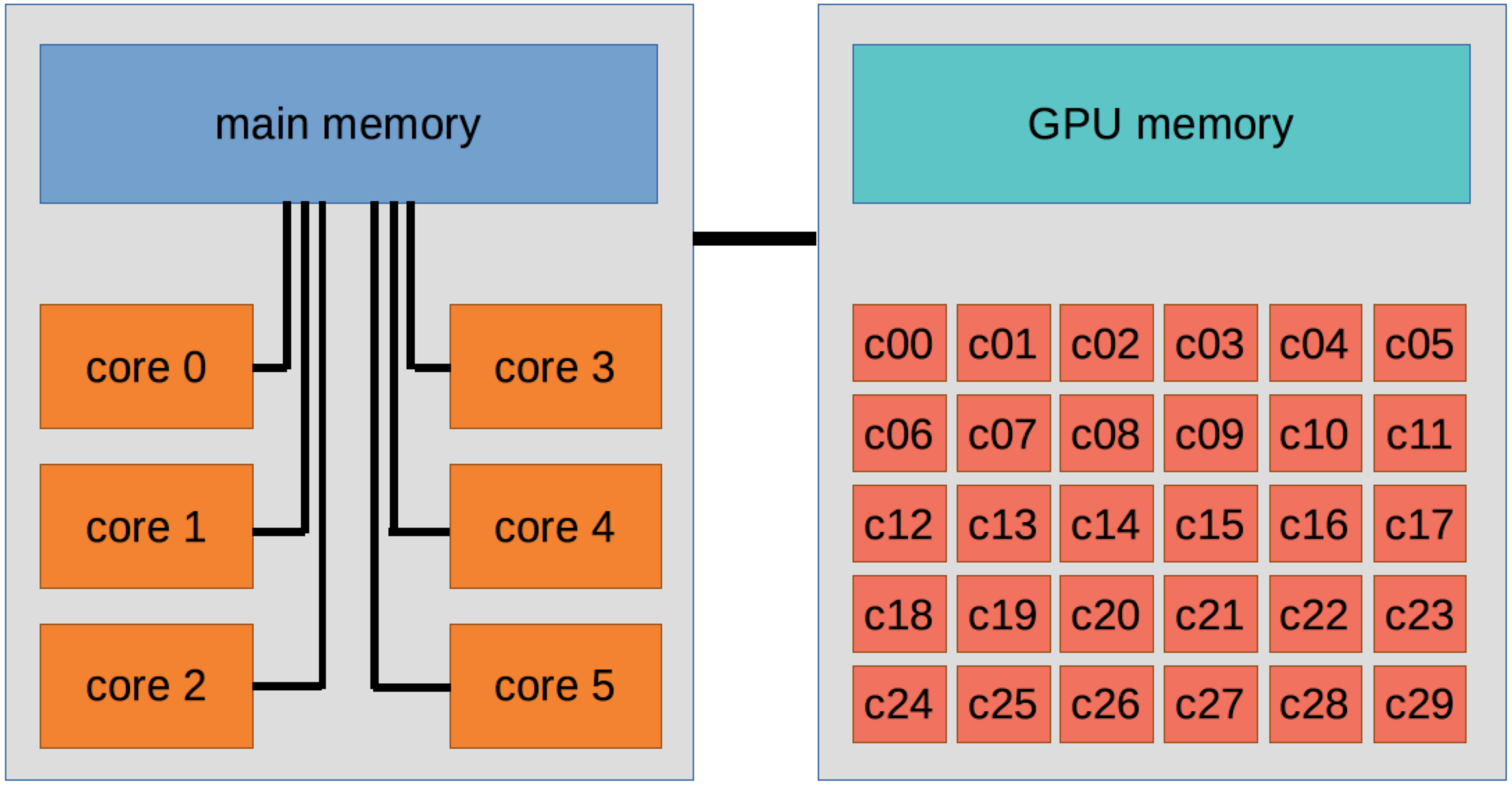
Philipp Girichidis

February 04, 2020

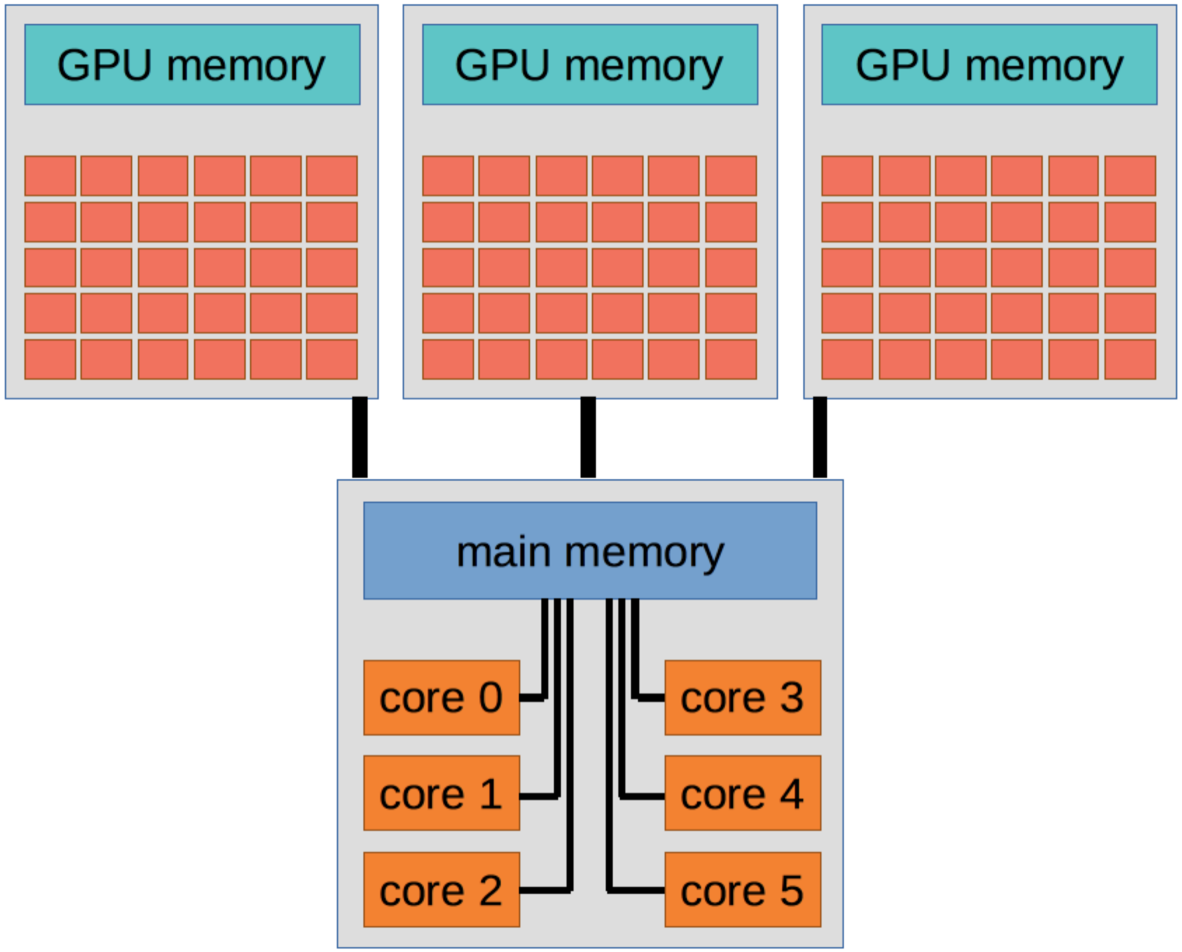
Outline

- Hardware architecture
- serial vs. parallel computing
- OpenMP parallelisation
- MPI parallelisation
- mixed OpenMP and MPI
- perfectly parallelizing algorithms
- hydrodynamics and domain decomposition
- long-range forces and communication

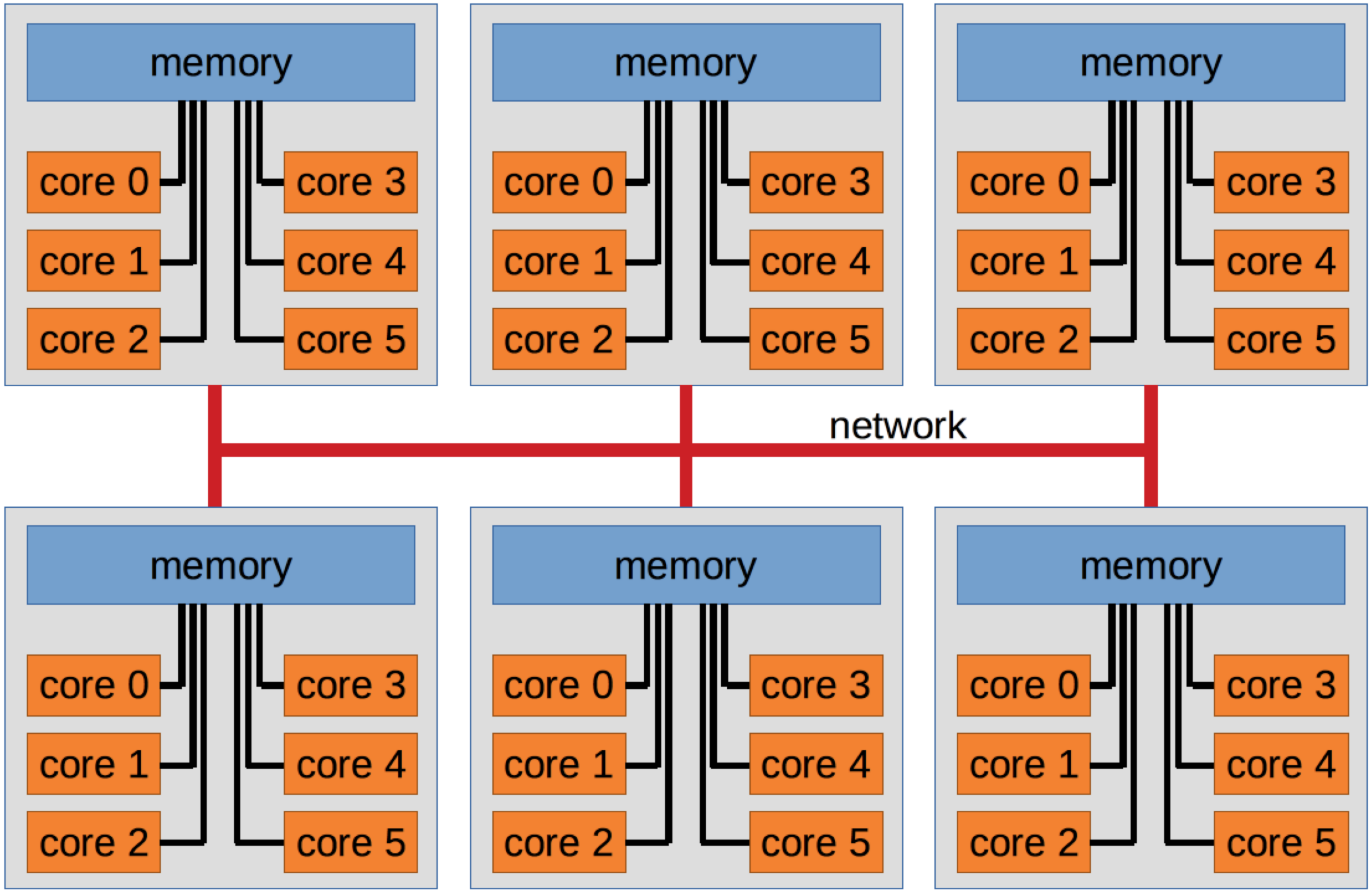
CPU and GPU


main memory
core 0
core 1
core 2
core 3
core 4
core 5
GPU memory
c00
c01
c02
c03
c04
c05
c06
c07
c08
c09
c10
c11
c12
c13
c14
c15
c16
c17
c18
c19
c20
c21
c22
c23
c24
c25
c26
c27
c28
c29

CPU with multiple GPUs



CPU network



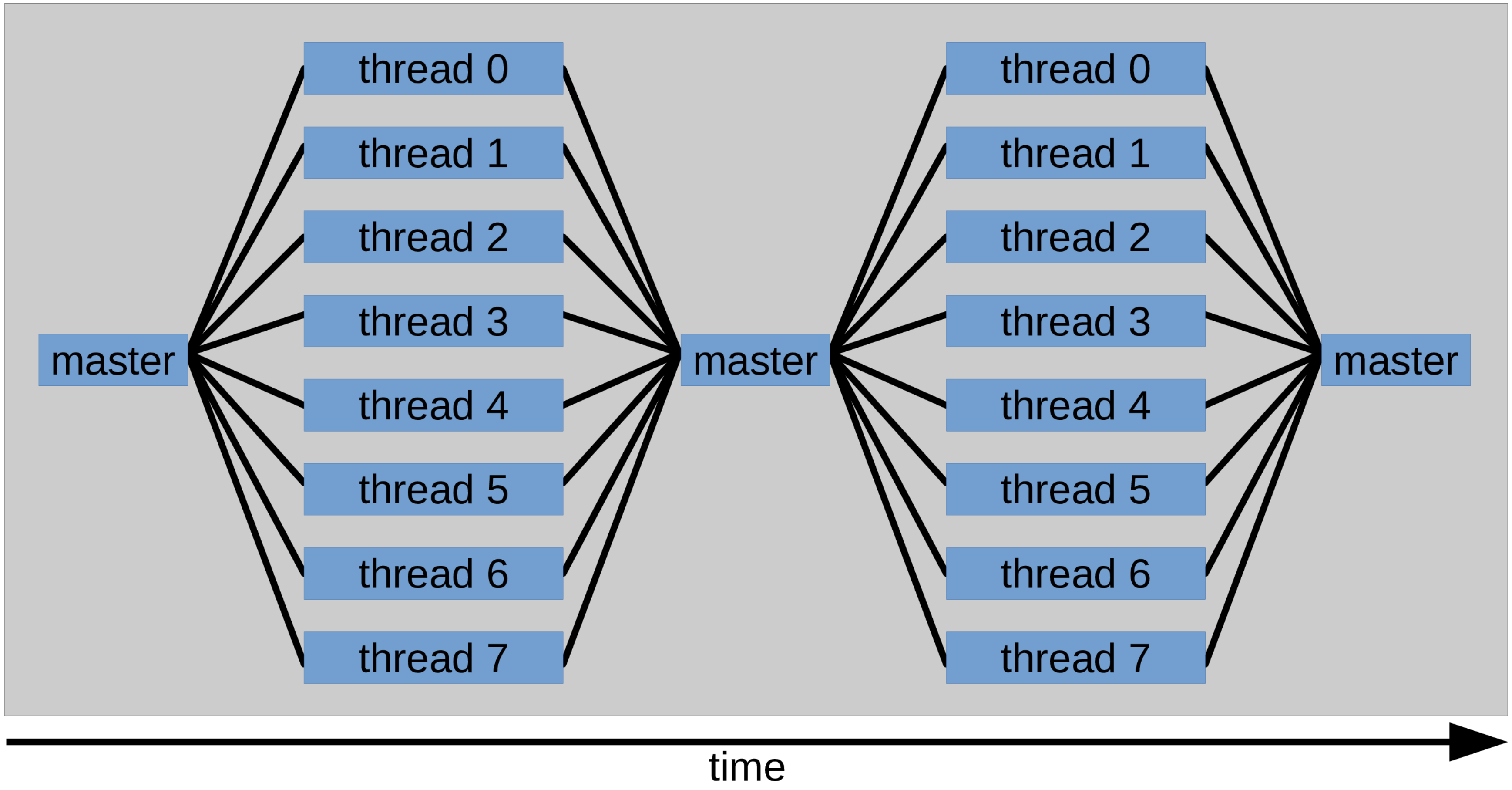
Why parallel computing

- multiple cores are faster
  - if algorithm parallelises well
  - if communication is fast
- data does not fit on memory
- different tasks on different hardware
  - complex instructions on CPU
  - simple code on GPU

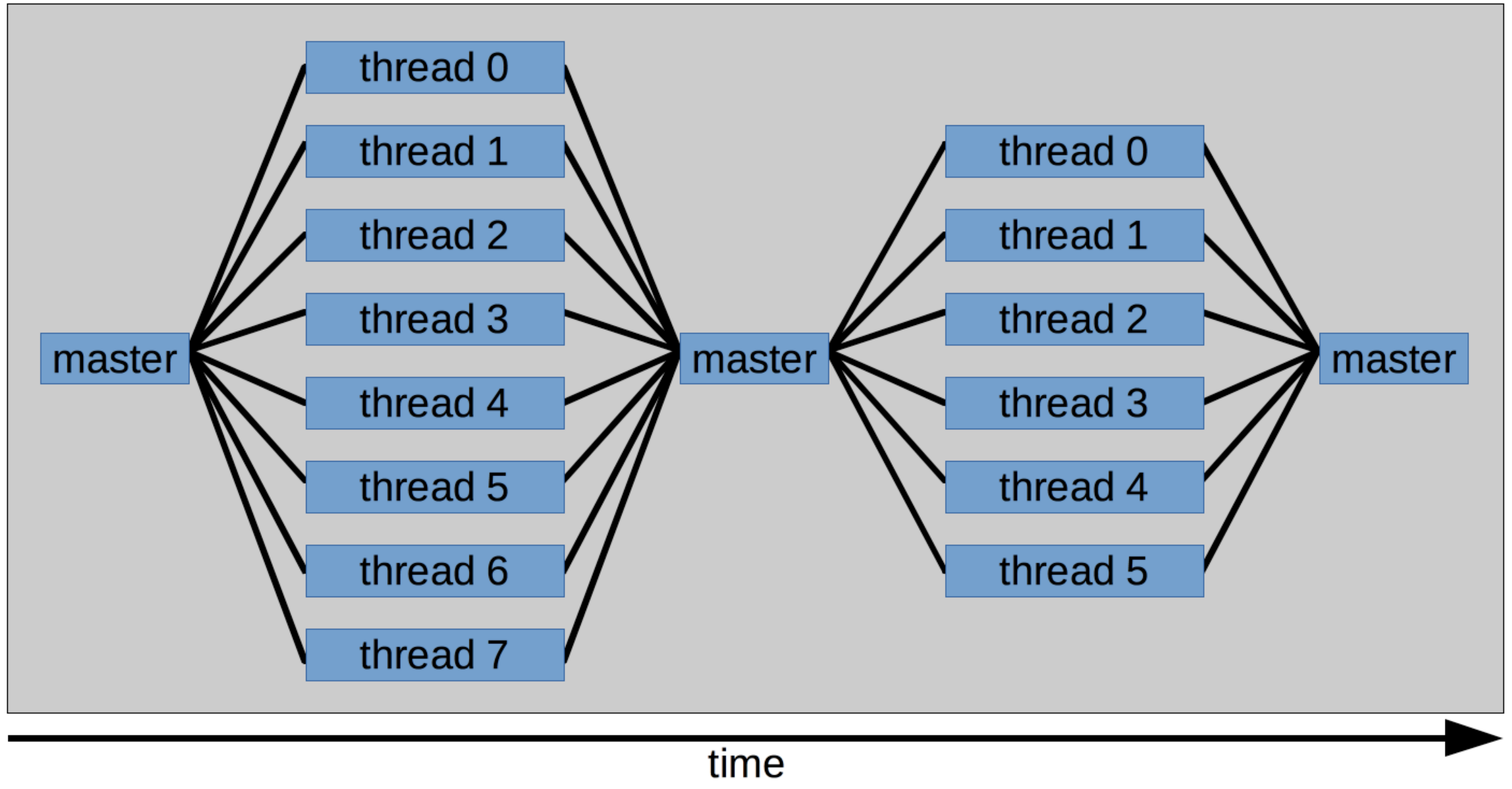
OpenMP parallelization

- shared memory parallelization
- one global process on the machine
- temporally occupies multiple cores/threads
- pro: simpler coding
- cons: limited to one node / CPU unit

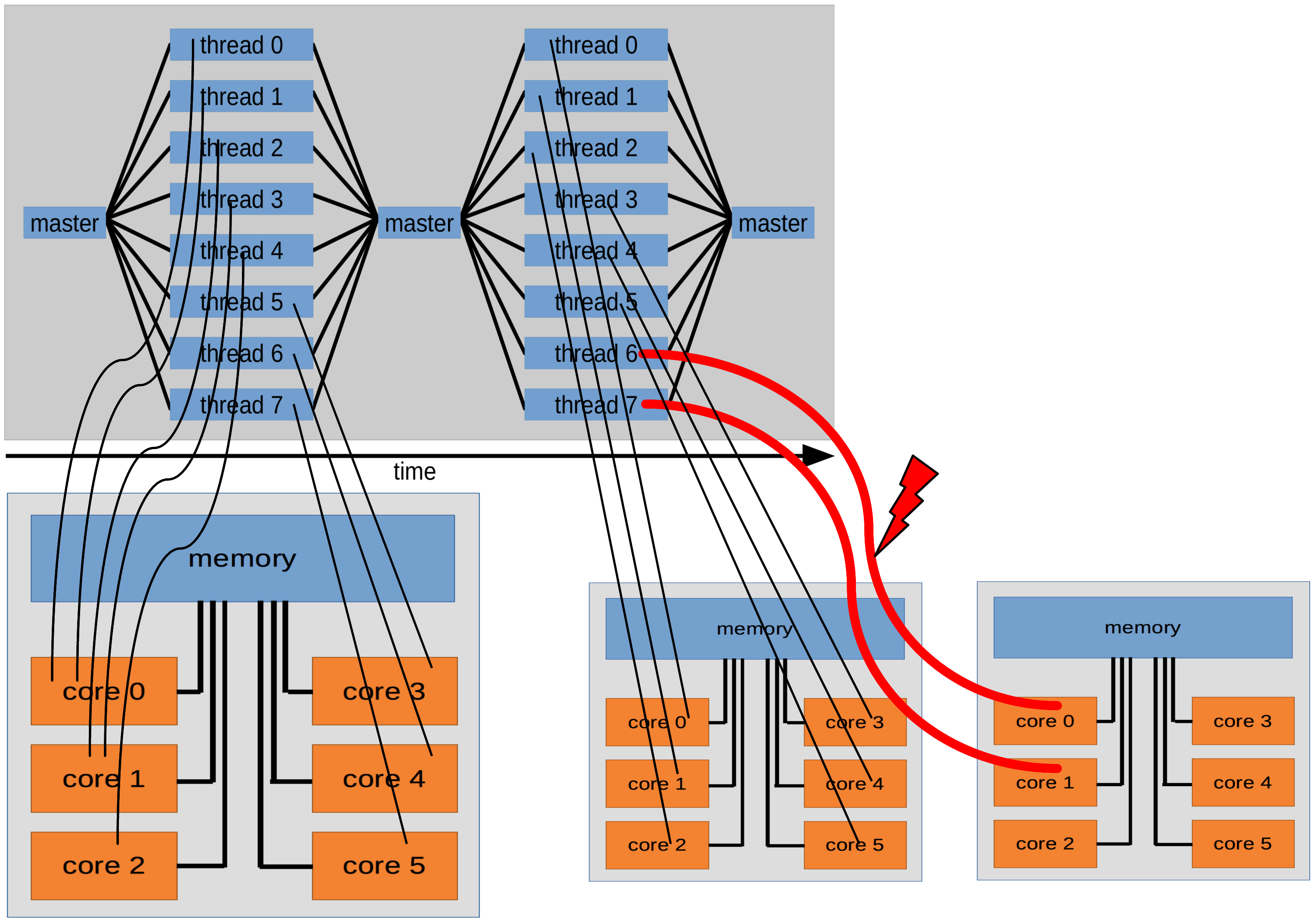
OpenMP parallelization



OpenMP parallelization



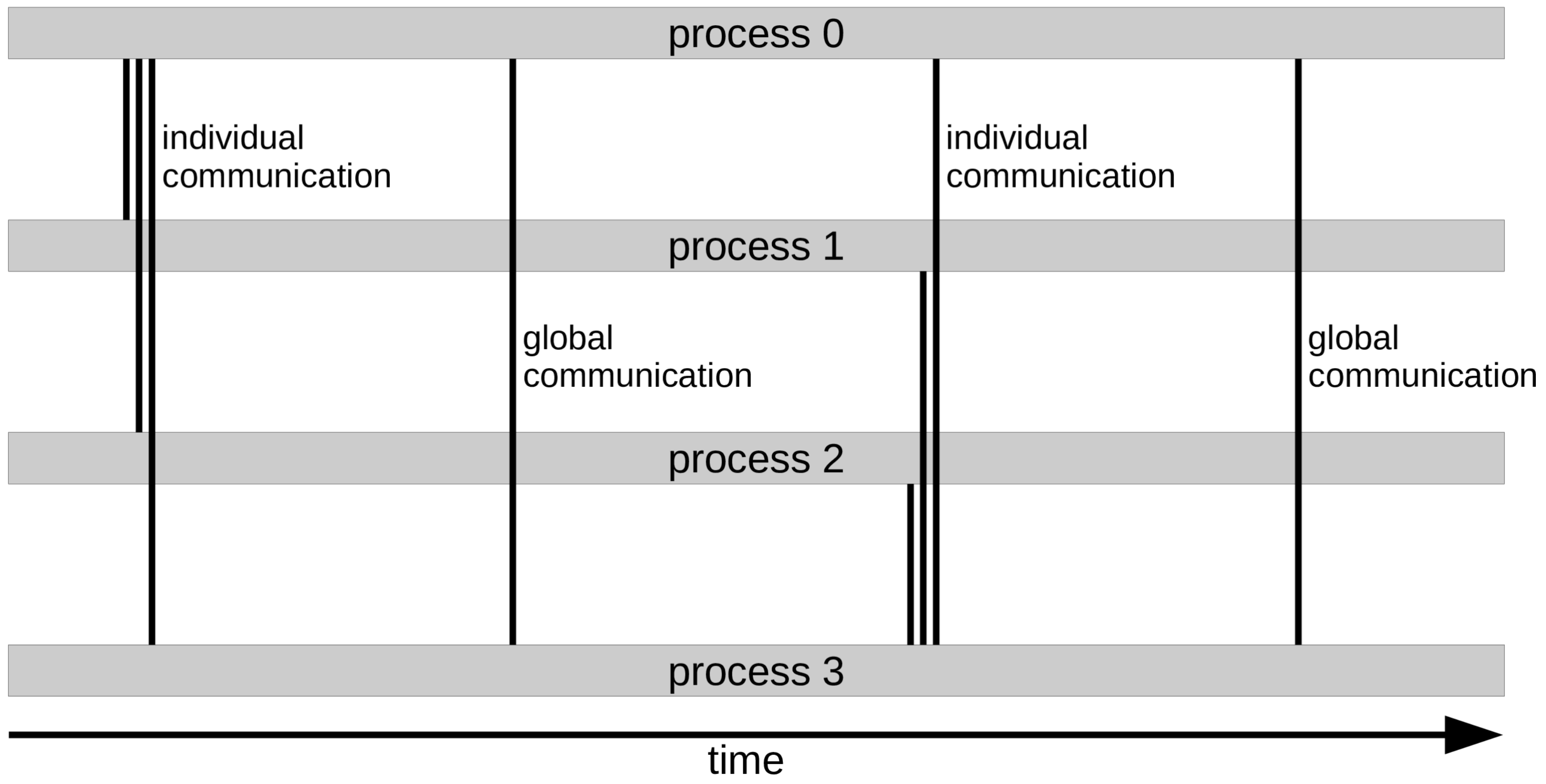
OpenMP parallelization



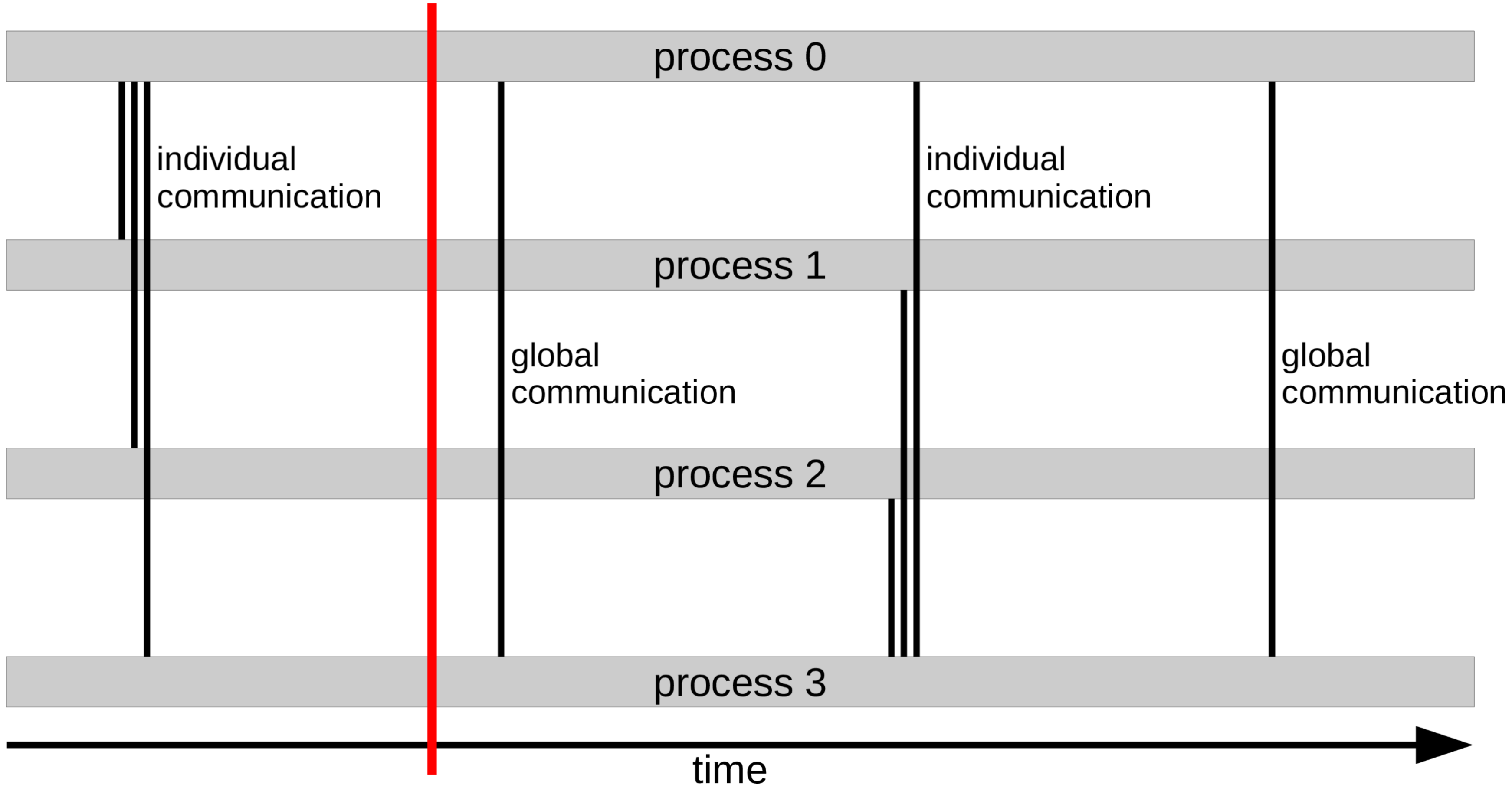
MPI parallelization

- *distributed* memory parallelization
- multiple individual OS processes
- in priniple independent execution
- occupies multiple cores
- every process own part of memory
- processes need to communicate
- more complicated coding
- no limitations on *local* memory and *local* number of cores

MPI parallelization



MPI parallelization



MPI barrier: wait until all processes are here

basic structure of MPI program

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {

    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Print off a hello world message
    printf("Hello world from processor %s, rank %d out of %d processors\n",
           processor_name, world_rank, world_size);

    // Finalize the MPI environment.
    MPI_Finalize();
}
```

MPI communication

```
int main(int argc, char** argv) {

    ...

    // send data to other processor
    MPI_Send(data, count, datatype, destination, tag, MPI_communicator);

    // receive data from other processor
    MPI_Recv(data, count, datatype, source, tag, MPI_communicator, status);

    ...
}
```

MPI communication II

```
int main(int argc, char** argv) {

    ...
    // initialise 2 MPI processes

    if(world_rank == 0)
    {
        // send data to other processor
        MPI_Send(data, count, datatype, destination=1, tag, MPI_comm);
    }
    else // world_rank == 1
    {
        // receive data from other processor
        MPI_Recv(data, count, datatype, source=0, tag, MPI_comm, status);
    }

    ...
}
```

MPI communication III

```
int main(int argc, char** argv) {

    ...
    // initialise N MPI processes

    if(world_rank == 0){
        for(int i=1; i<N; i++)
        {
            // send data to other processor
            MPI_Send(data, count, datatype, destination=i, tag, MPI_comm);
        }
    }
    else
    {
        // receive data from other processor
        MPI_Recv(data, count, datatype, source=0, tag, MPI_comm, status);
    }

    ...
}
```

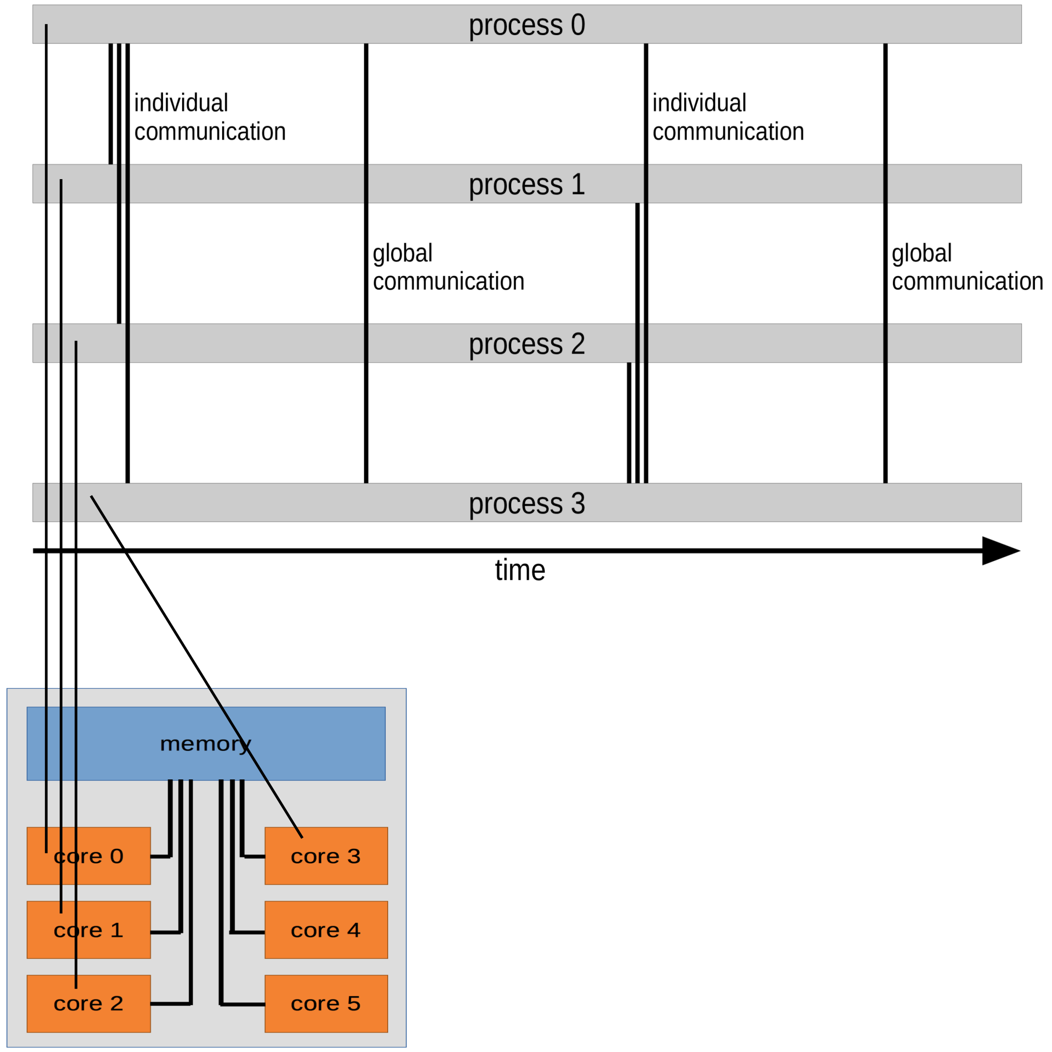
MPI communication IV

```
int main(int argc, char** argv) {

    ...
    // distribute to all

    if(world_rank == 0){
        // send data to other processor
        MPI_Bcast(data, count, datatype, tag, MPI_comm);
    }
    else
    {
        // receive data from other processor
        MPI_Recv(data, count, datatype, source=0, tag, MPI_comm, status);
    }

    ...
}
```

MPI other commands

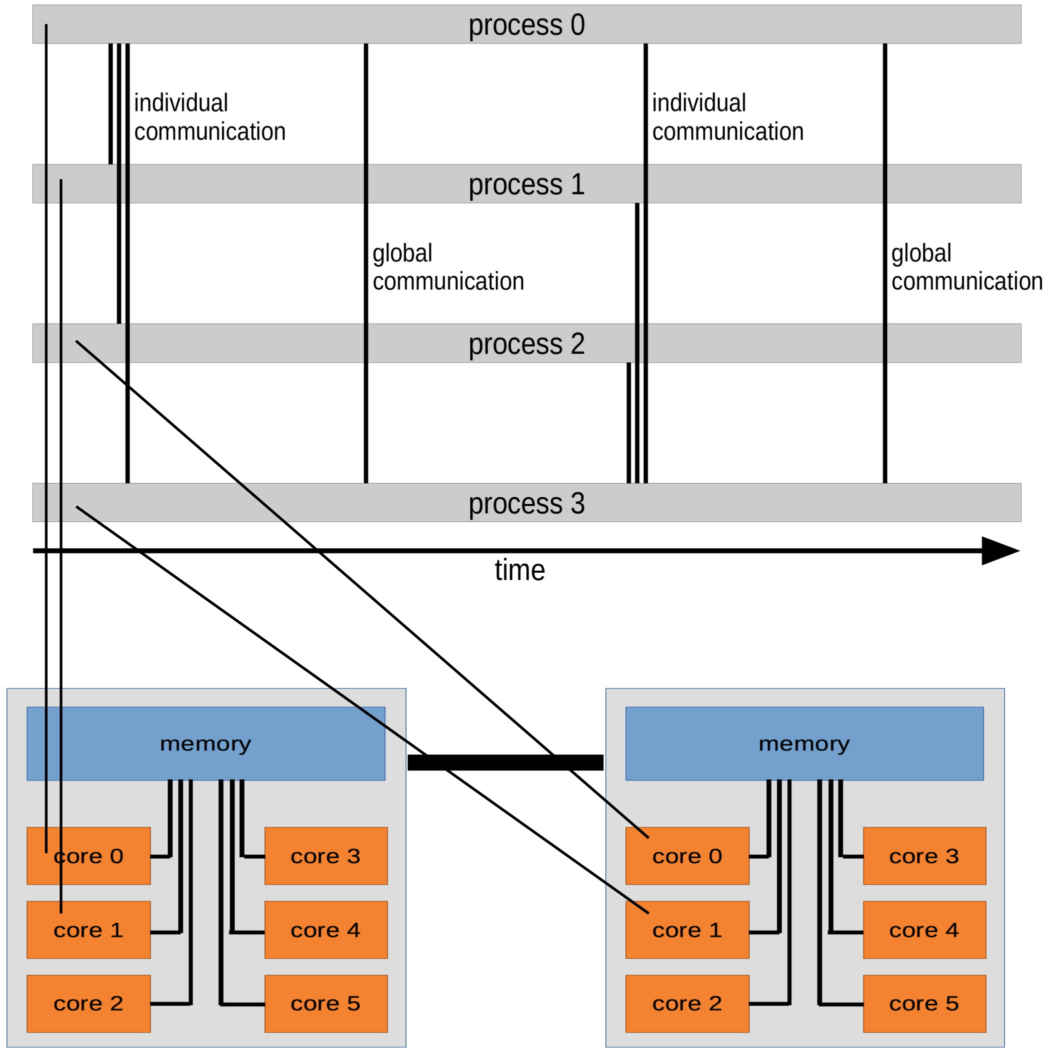
```
int main(int argc, char** argv) {

    ...
    // wait here
    MPI_Barrier(...)

    // collect from all
    MPI_Gather(...)

    // reduce
    MPI_Reduce(..., mode=MODE)
    ...
}

MODE:
MPI_MAX  : find minimum
MPI_MIN  : find maximum
MPI_SUM  : sum all values
MPI_PROD : multiply all values
MPI_LAND : logical and
MPI_LOR  : logical or
....
```

MPI parallelization



option 1
- simple pinning
- each process: one core

MPI parallelization



option 2
- simple pinning
- processes distributed onto several nodes
- 2/6 cores occupied
- each process 1/2 memory

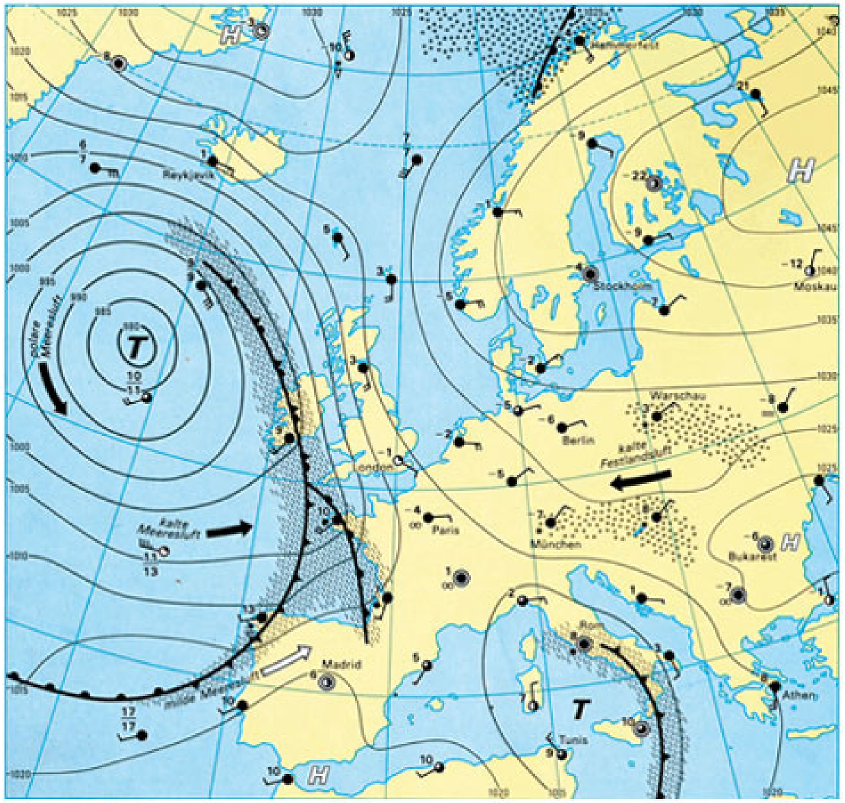
mixed OpenMP and MPI

- start MPI process on every node
- inside node OpenMP with shared memory

Perfect parallelization

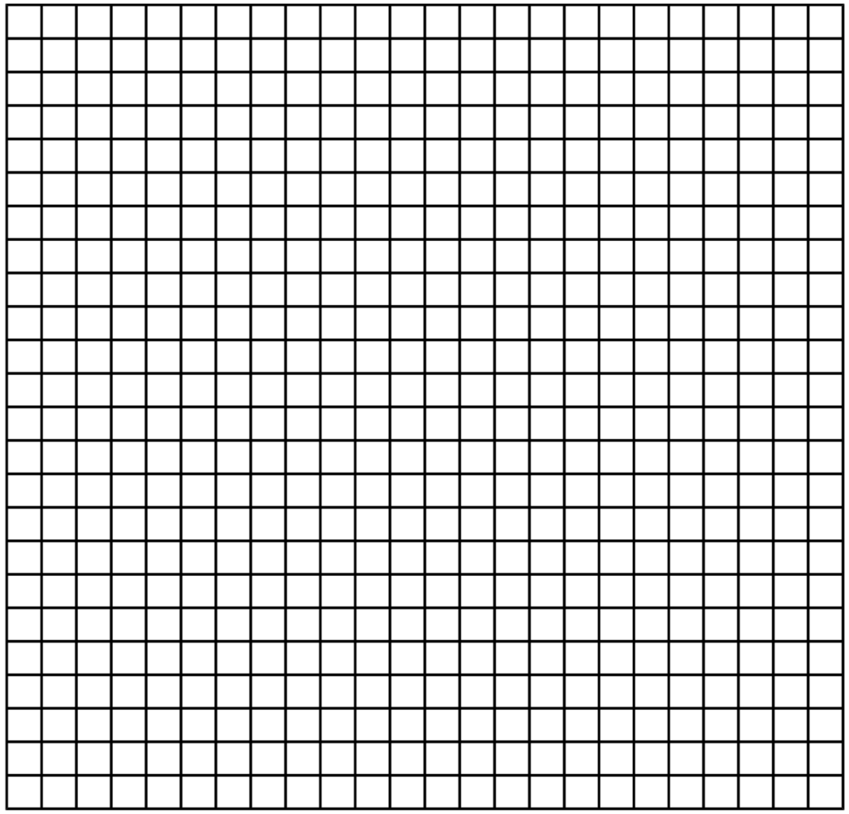
- vector multiplication
- simple matrix operations
- Monte Carlo simulations
- independent parameter scan

hydrodynamics everywhere



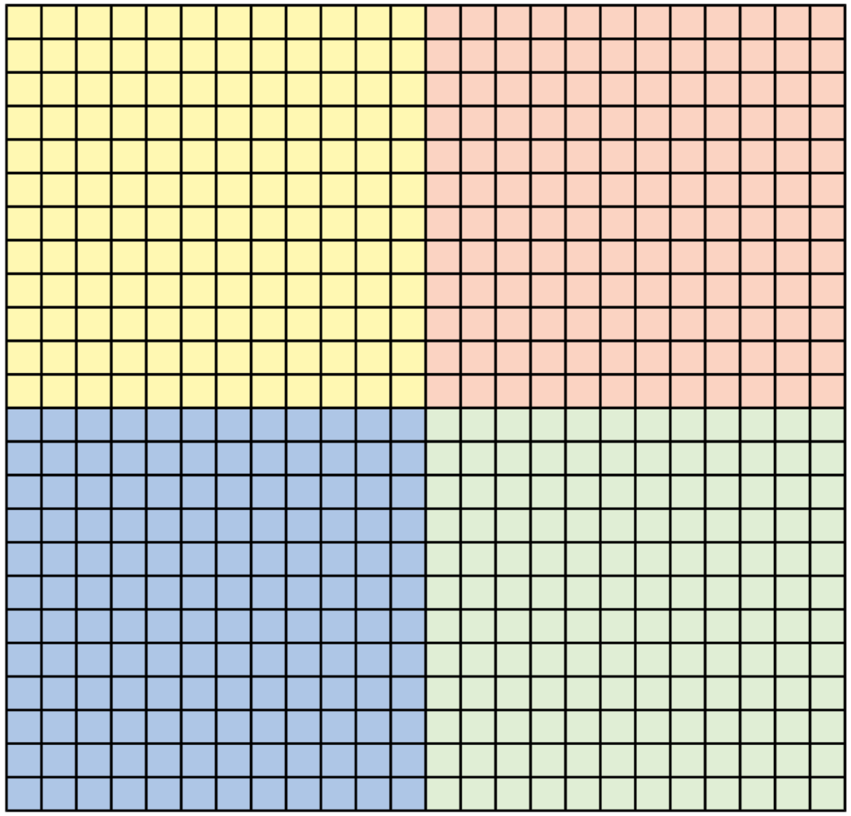
hydrodynamics

- Solve discretized fluid equations on a grid
- simplest case: uniform, periodic grid



hydrodynamics

- Solve discretized fluid equations on a grid
- simplest case: uniform, periodic grid
- split domain between processors



hydrodynamcis - equations

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0$$

$$\frac{\partial \rho \mathbf{v}}{\partial t} + \nabla \cdot \left[\rho \mathbf{v} \mathbf{v}^{\mathrm{T}} + \underbrace{\left(P_{\mathrm{th}} + \frac{||\mathbf{B}||^2}{8\pi} \right)}_{P_{\mathrm{tot}}} \mathsf{I} - \frac{\mathbf{B}\mathbf{B}^{\mathrm{T}}}{4\pi} \right] = \rho \mathbf{g}$$

$$\frac{\partial e}{\partial t} + \nabla \cdot \left[\left(u + \frac{\rho ||\mathbf{v}||^2}{2} + \frac{||\mathbf{B}||^2}{8\pi} + \frac{P_{\mathrm{th}}}{\rho} \right) \mathbf{v} - \frac{\mathbf{B} \left(\mathbf{v} \cdot \mathbf{B} \right)}{4\pi} \right] = \rho \mathbf{v} \cdot \mathbf{g}$$

$$\frac{\partial \mathbf{B}}{\partial t} - \nabla \times (\mathbf{v} \times \mathbf{B}) = 0$$

discretization

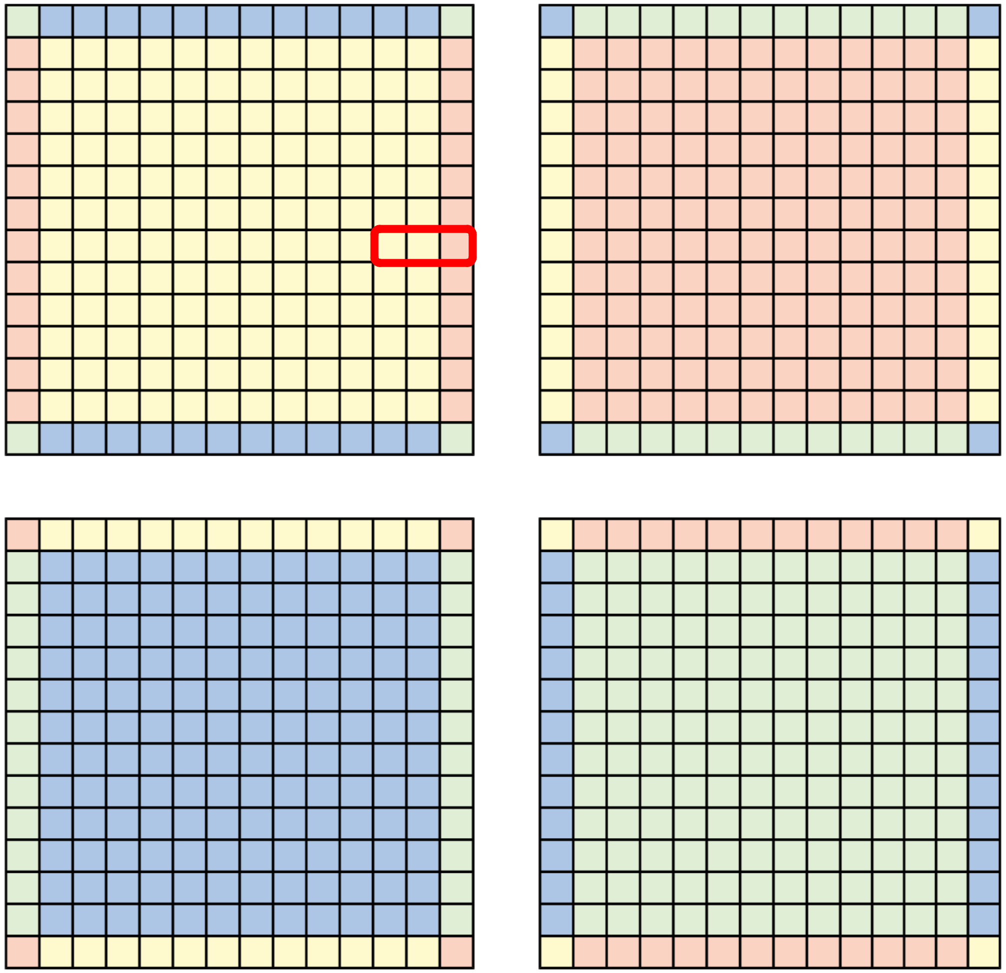
- temporal and spatial discretization

$$\frac{\partial y}{\partial t} \rightarrow \frac{y_i^{n+1} - y_i^n}{\Delta t}$$

$$\frac{\partial^2 y}{\partial x^2} \rightarrow \frac{y_{i+1}^n - 2y_i^n + y_{i-1}^n}{\Delta x^2}$$

- 3-point stencil: need one neighbour in each direction

need neighbour cells



how to communicate

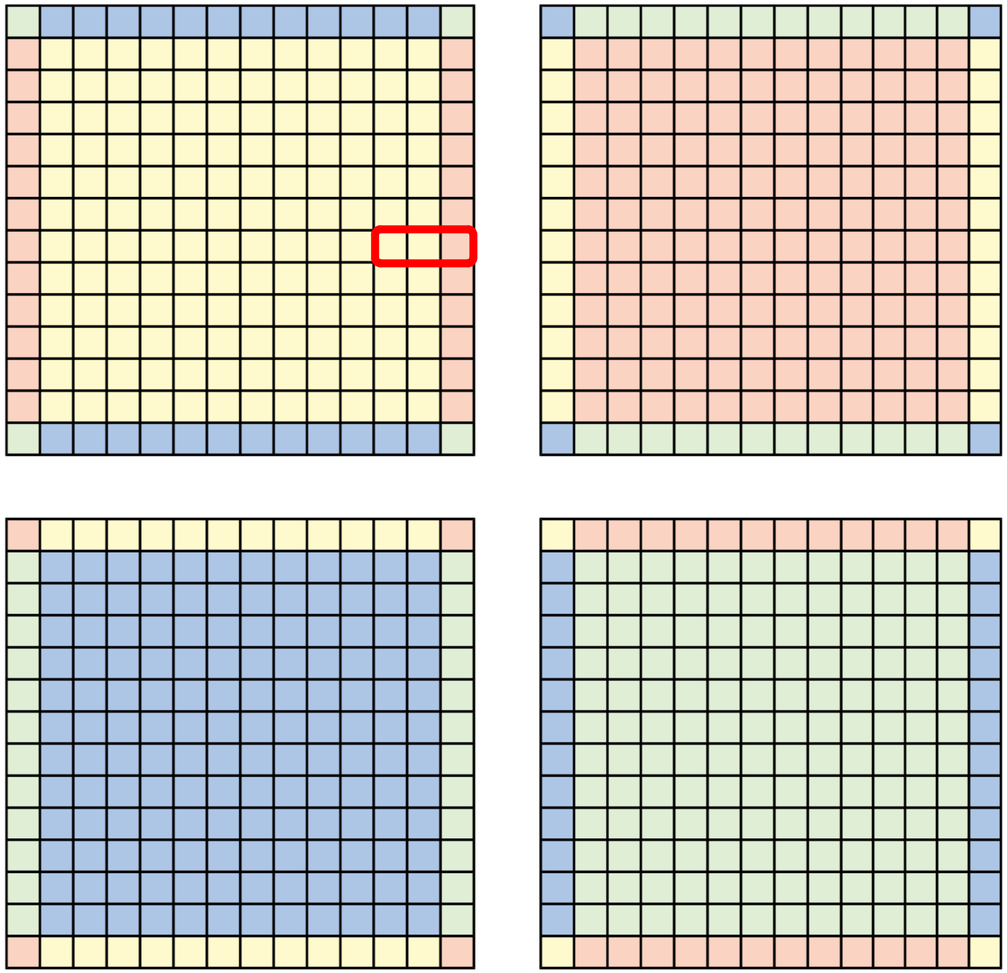
stupid way

```
do time_loop


  do x_loop
    do y_loop
      if at boundary
        # communicate
        MPI_get_neighbour()
      density  = ...
      momentum = ...
      energy   = ...
    done
  done

done
```

clever way

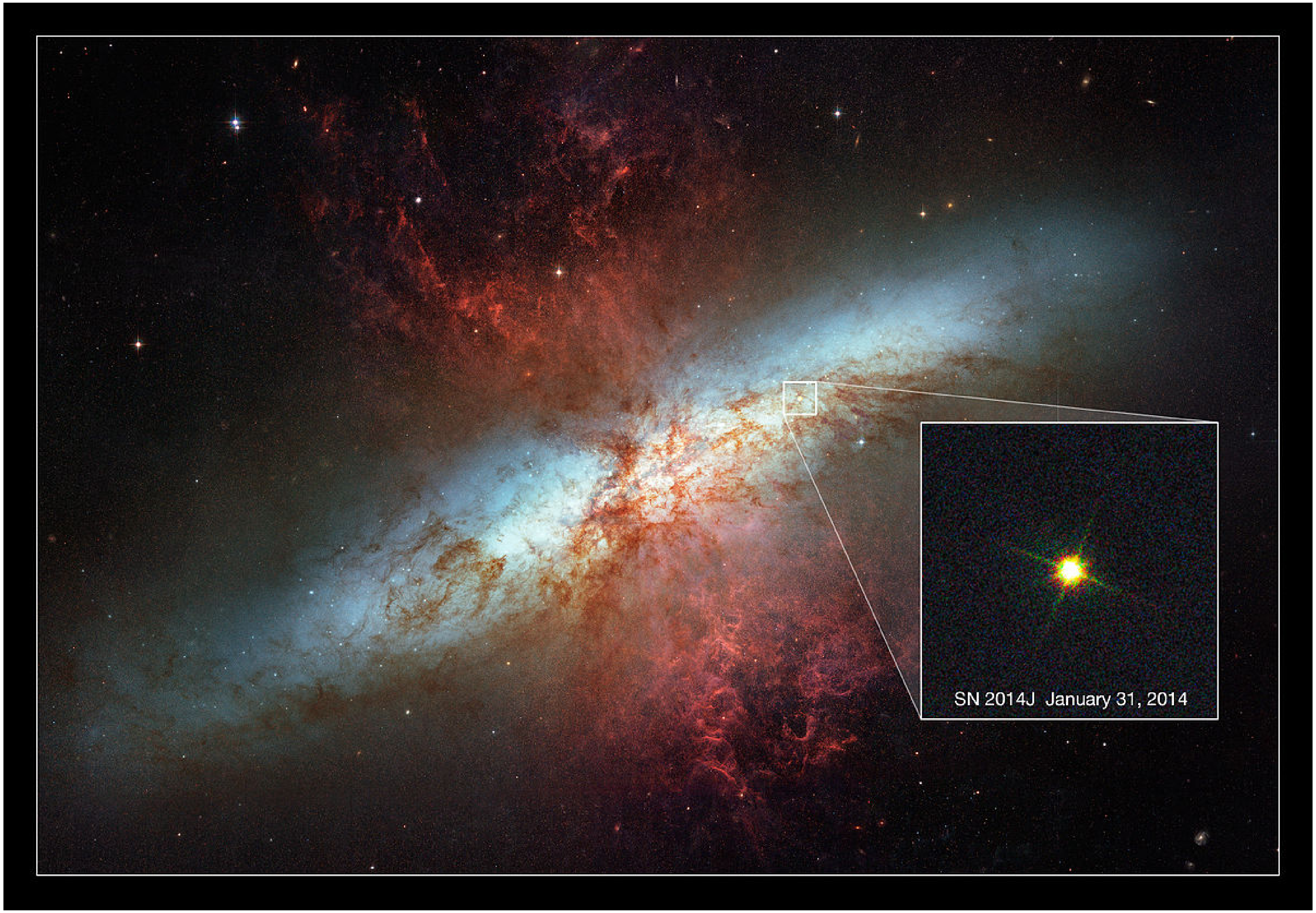
```
do time_loop
  # get a copy of neighbours
  MPI_get_neighbours()
  do x_loop
    do y_loop



        density  = ...
        momentum = ...
        energy   = ...
    done
  done
  MPI_send_neighbours()
done
```

guard (ghost) cells

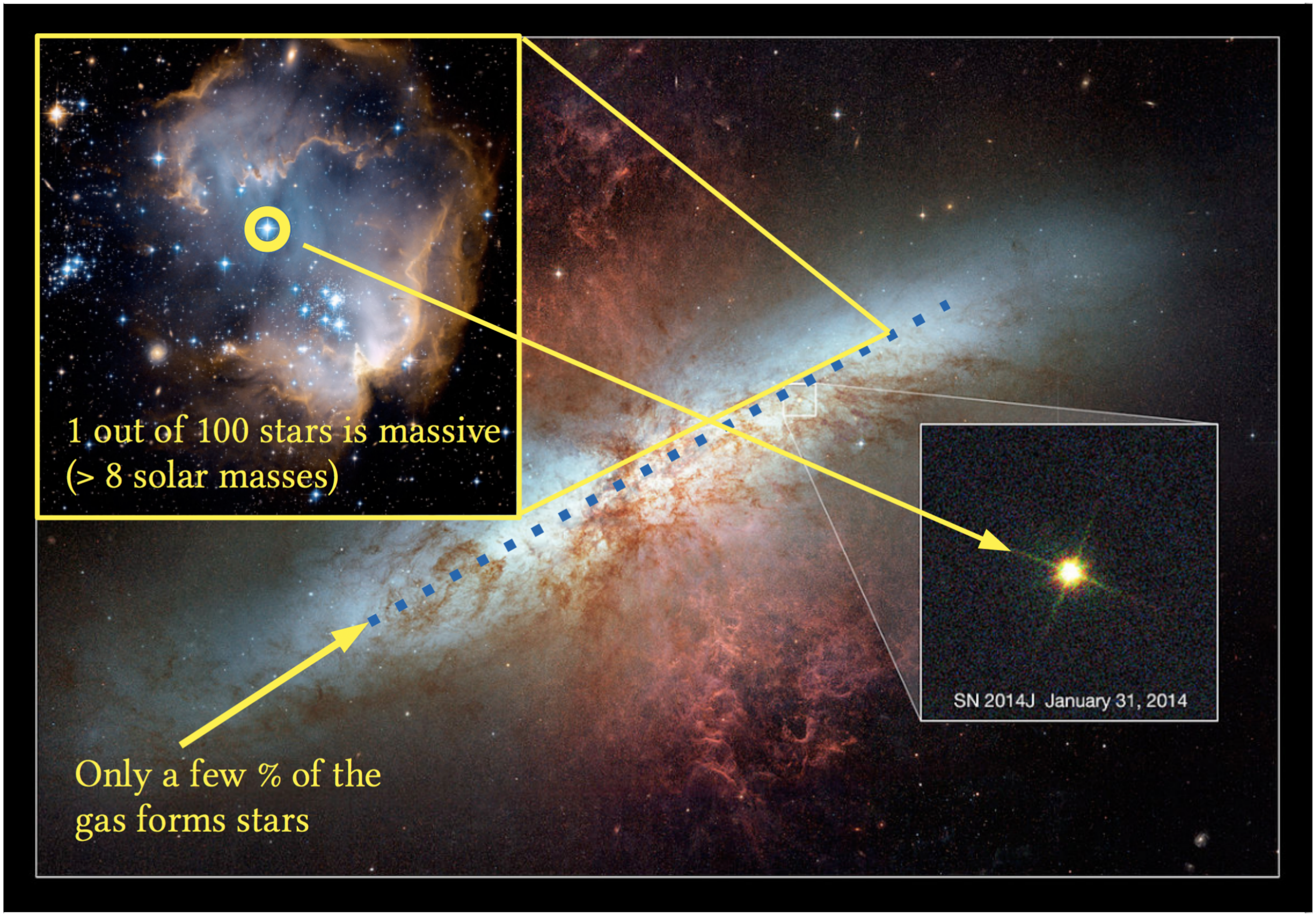


- use guard cells to reduce communication

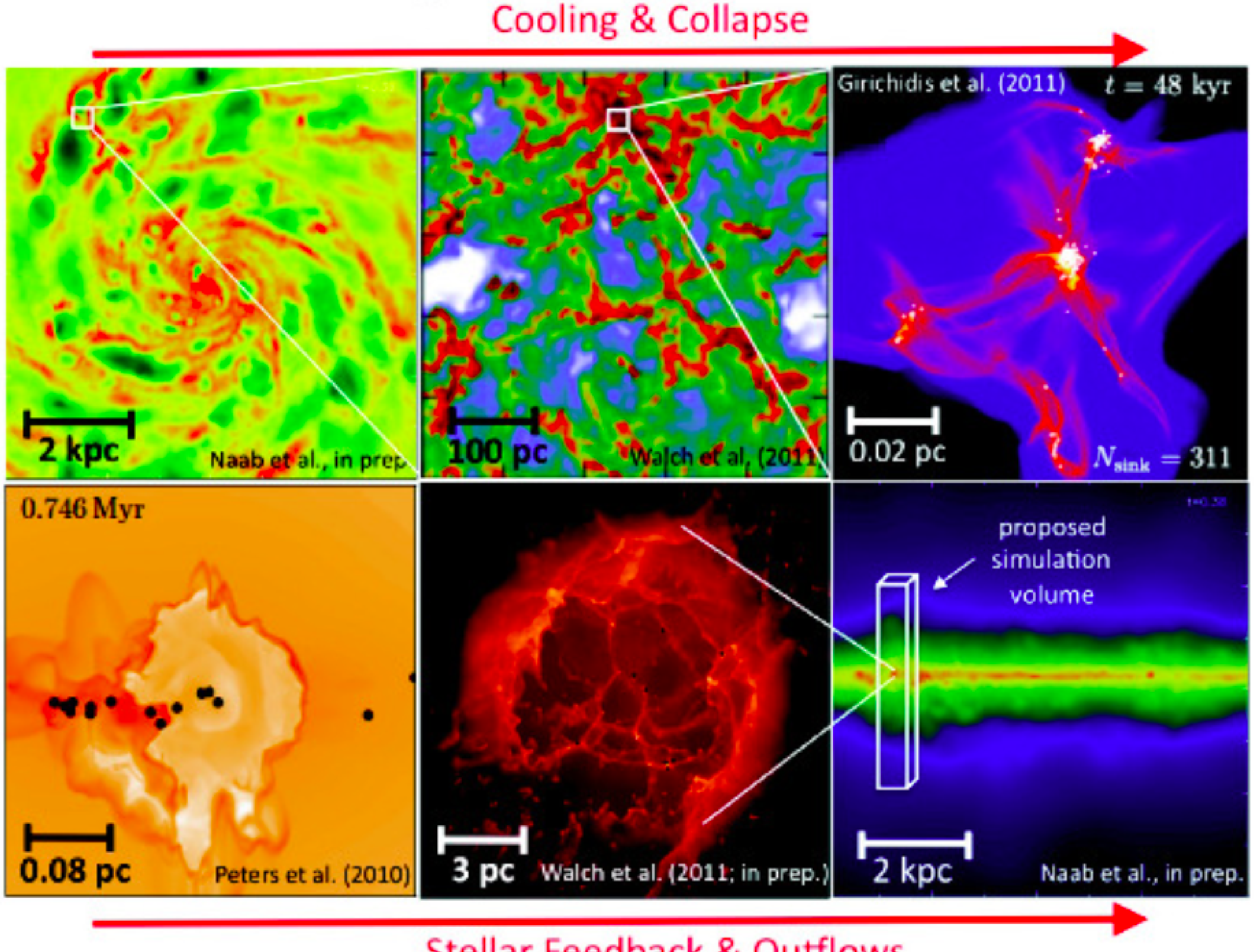
galaxy and ISM



galaxy and ISM

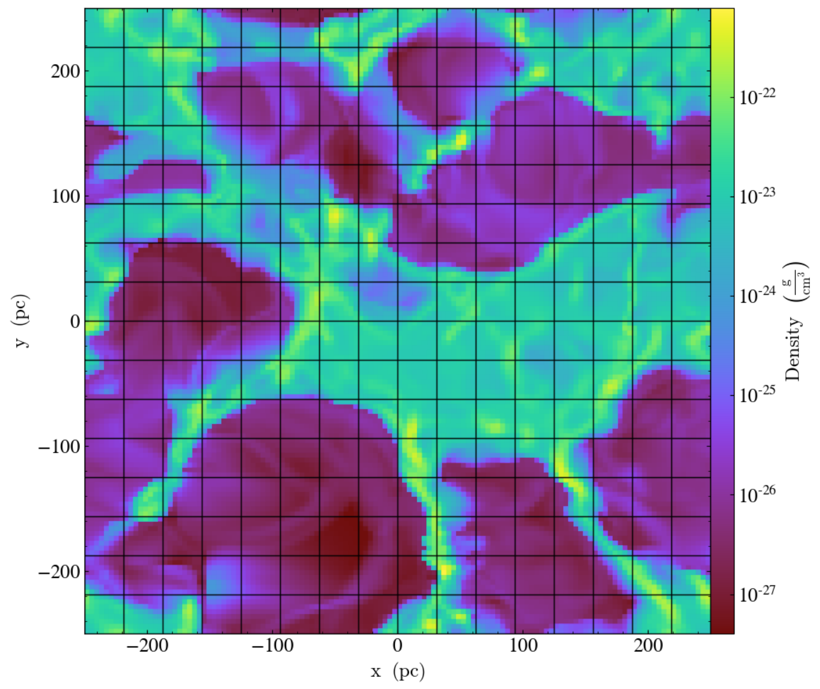

1 out of 100 stars is massive (> 8 solar masses)
SN 2014J January 31, 2014
Only a few % of the gas forms stars

Lifecycle of molecular clouds

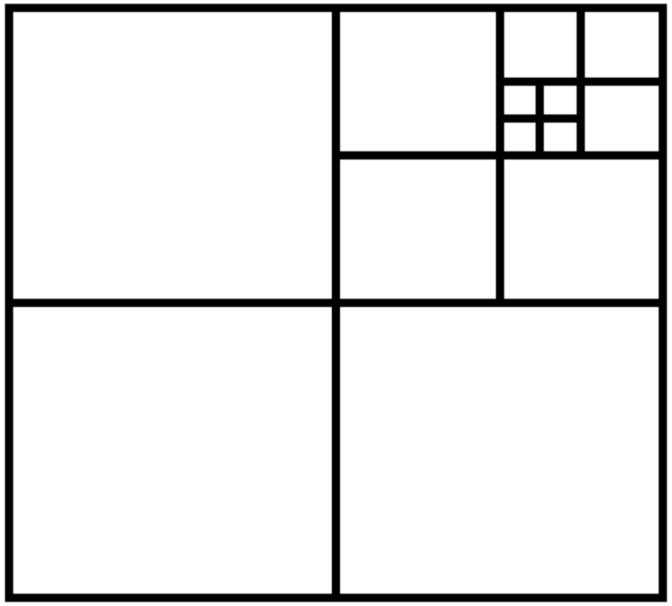


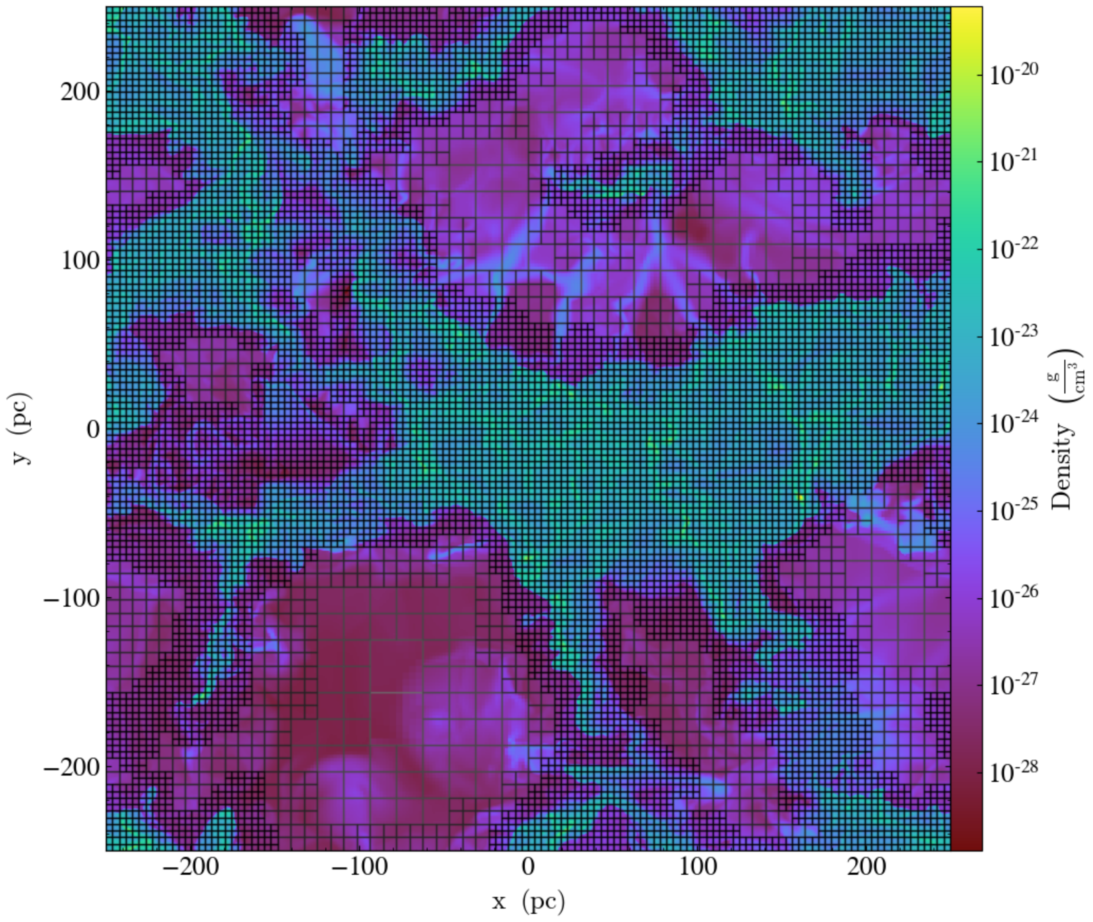
simulations of interstellar gas

- dense cold gas that forms clouds and stars
- diffuse warm gas
- hot gas that escapes the galaxy
- different scales (space/time), so need adaptive grid

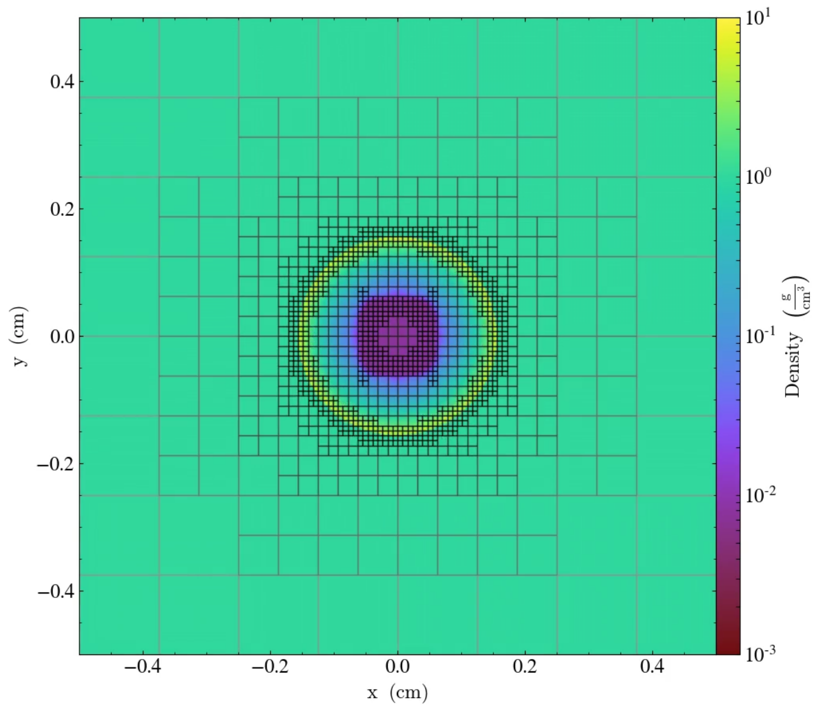


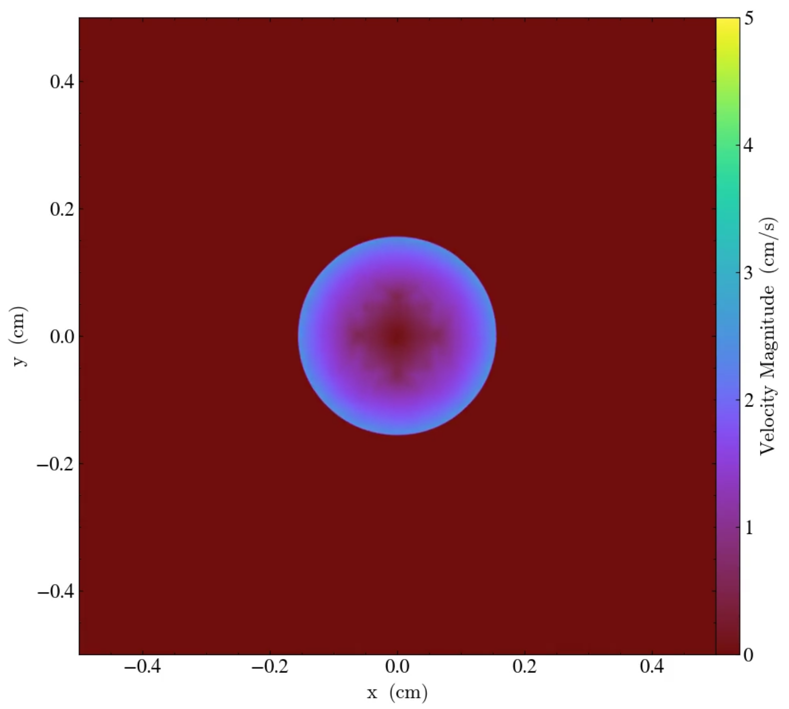
adaptive mesh refinement





example: Sedov explosion



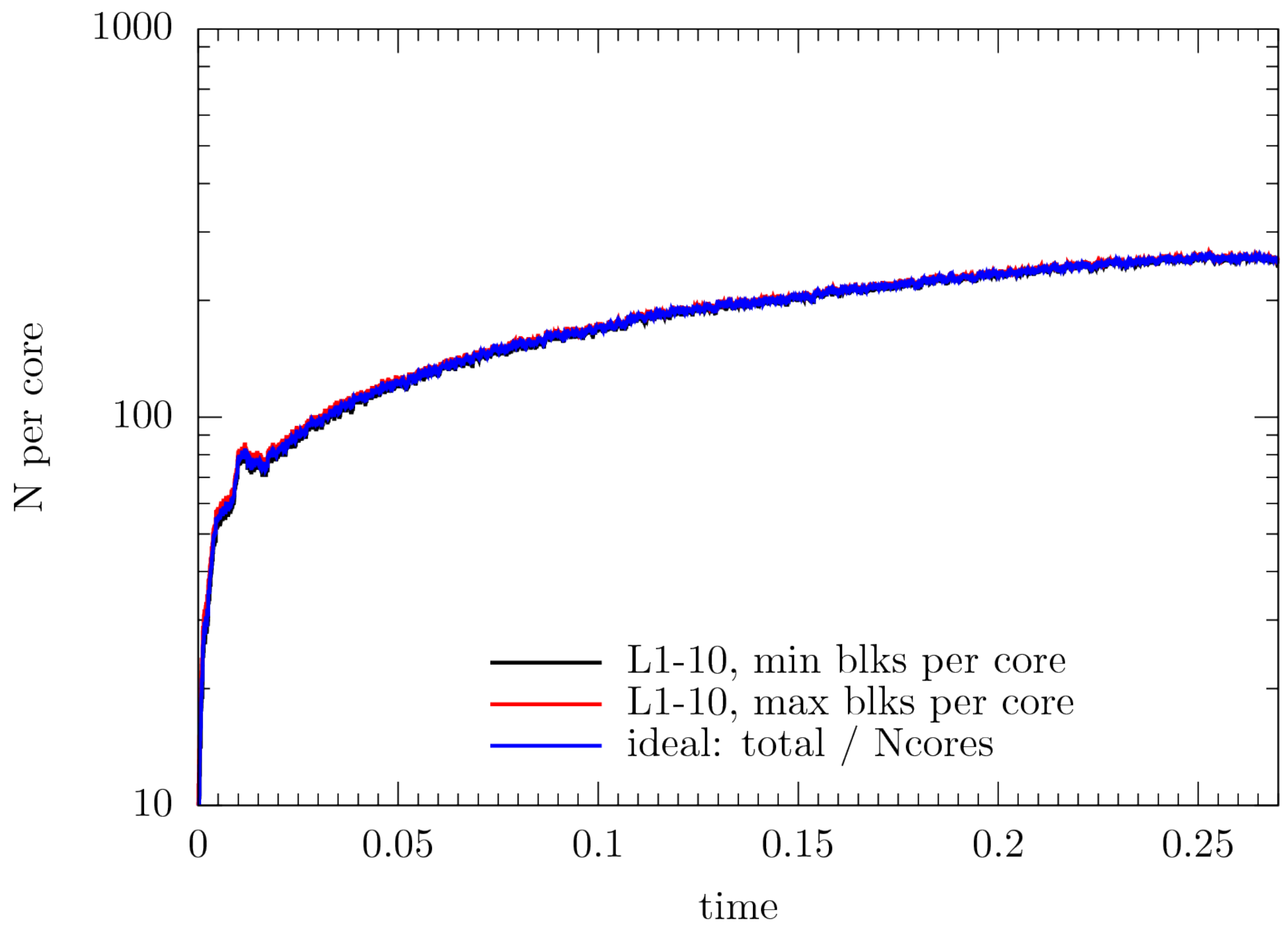


- dynamically follow the interesting gas structures (here shock)
- refine and derefine
- more complicated domain decomposition
- dynamical redistribution of regions between the cores

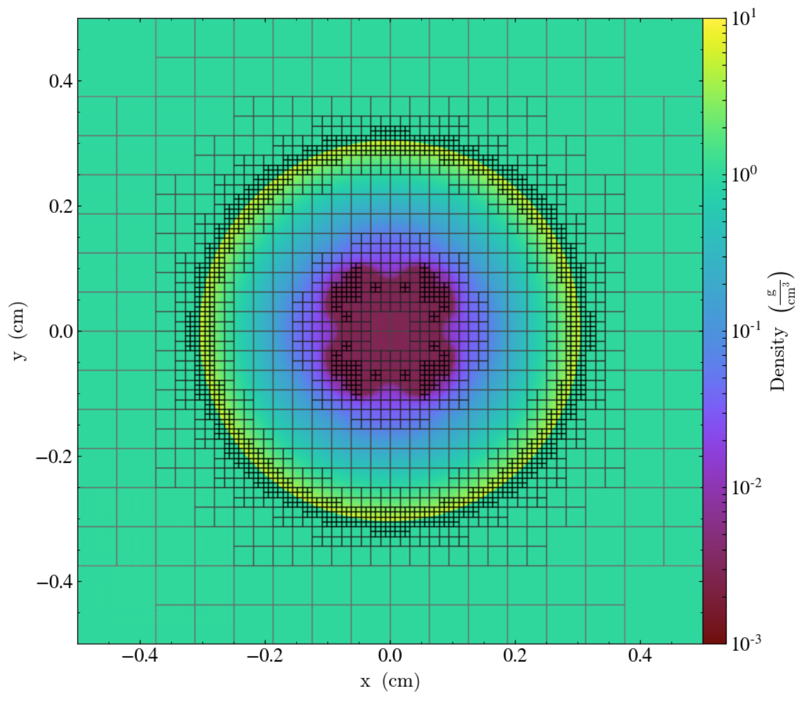
Domain decomposition

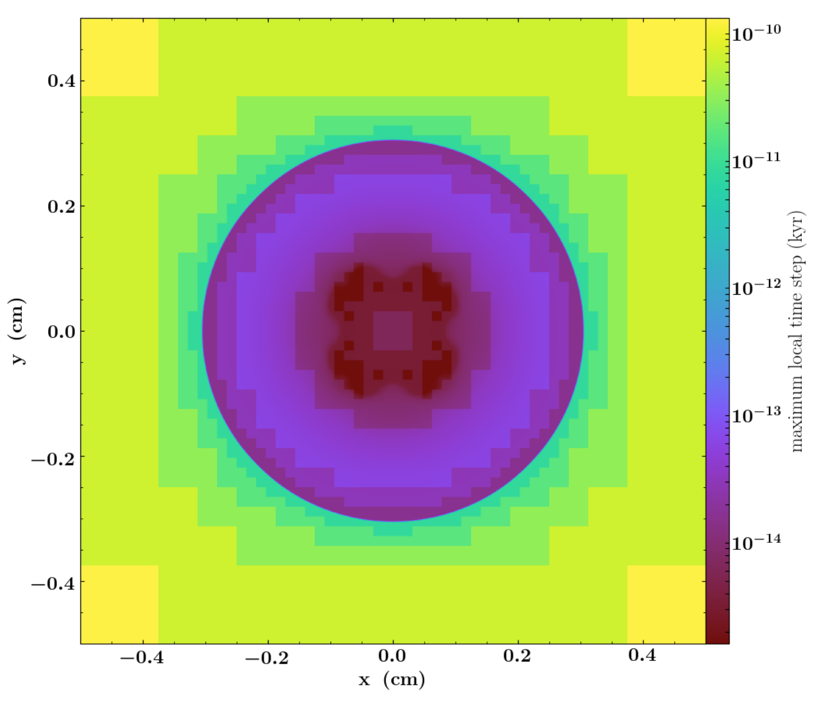
- simplest way:
  - each processor same number of cells
  - select domain with least communication (shortest border)
  - perfect memory distribution
- problematic if different cells require different cost
  - iterations depend on density, temperature
  - iterations depend on position

memory balancing



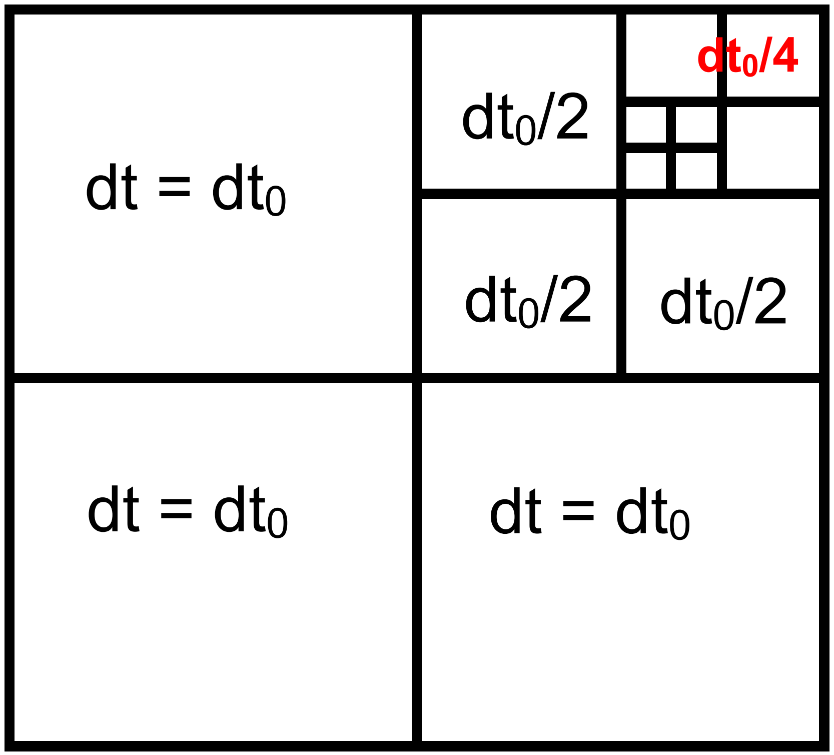
local time steps (actual work)





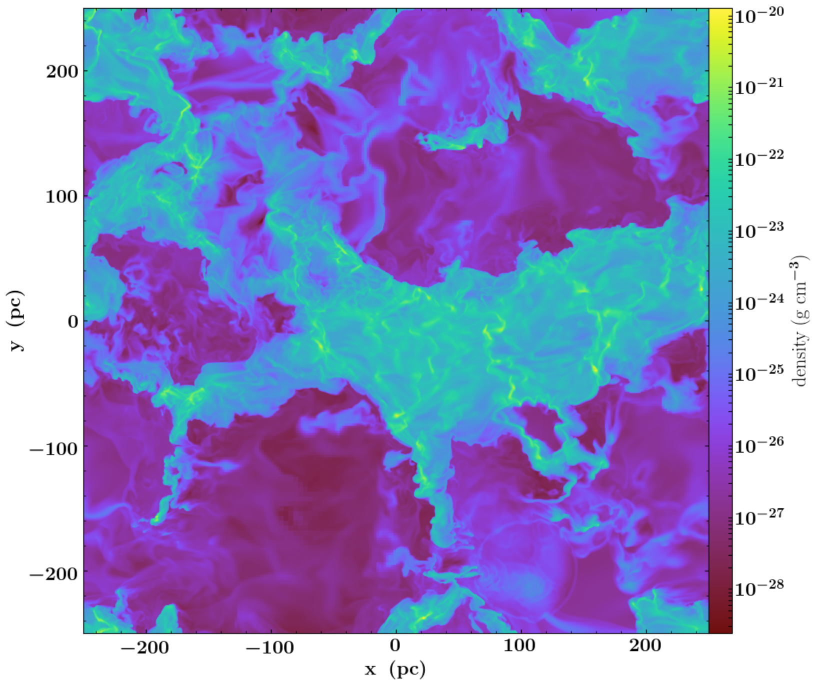
physics load balancing

- perfect distribution in memory
- but small cells interact on smaller time scales
- small blocks need to do more iterations
- cores with small cells do more work!

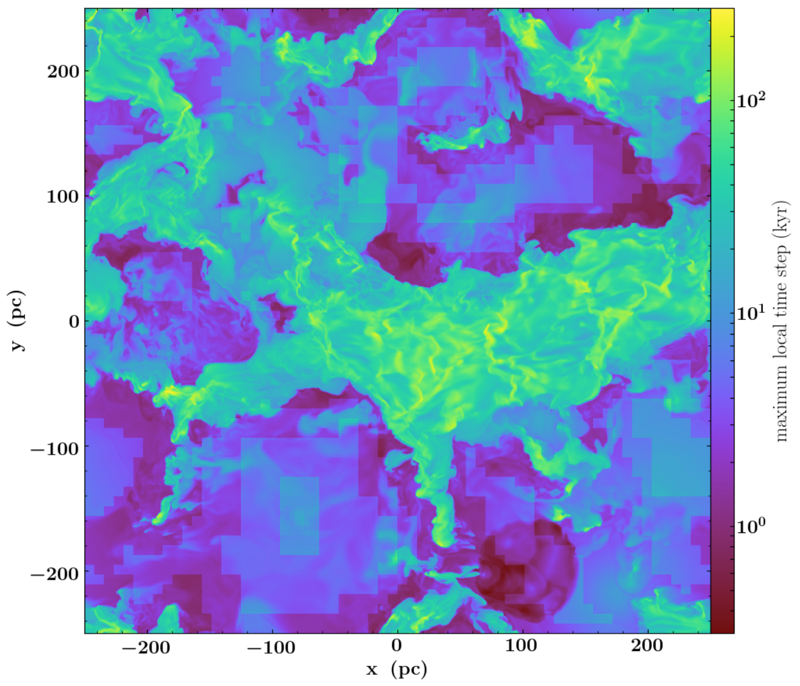


another example: ISM

density

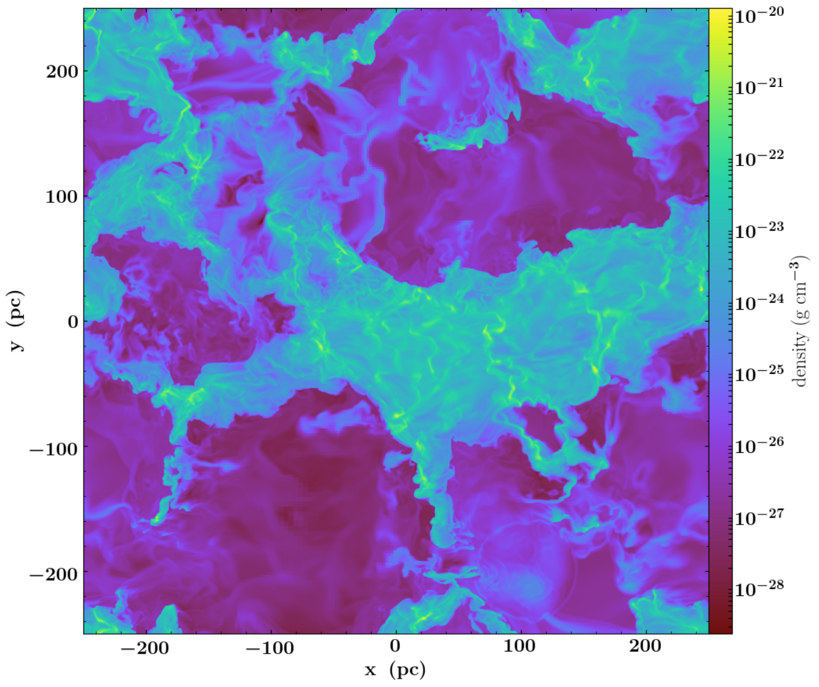


local hydro time

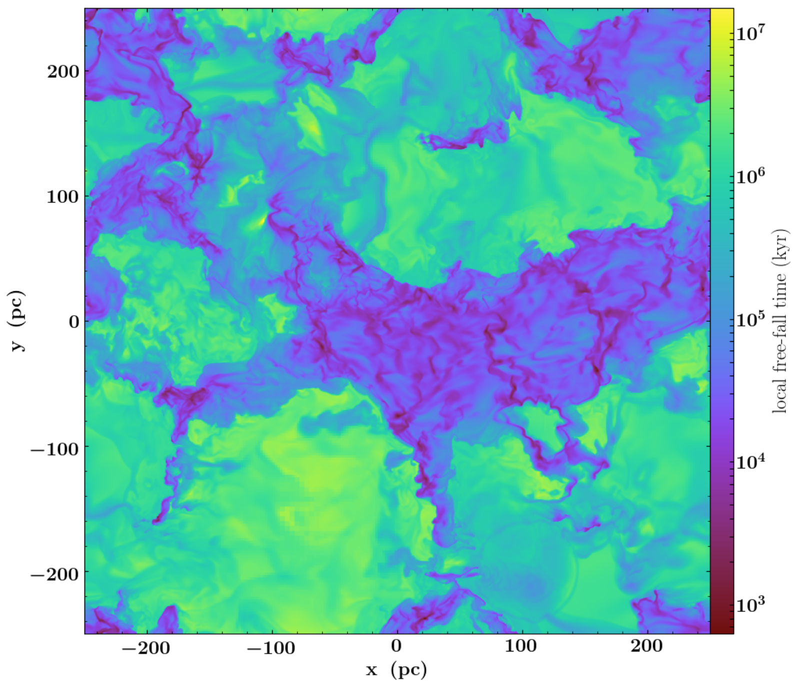


time scales

density

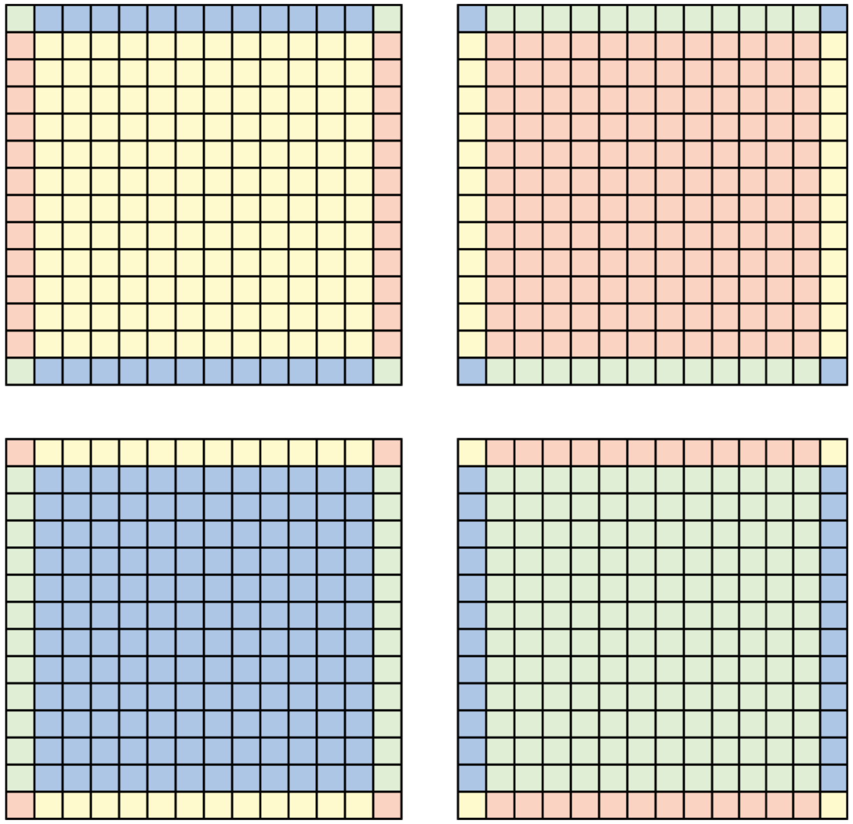


free-fall time

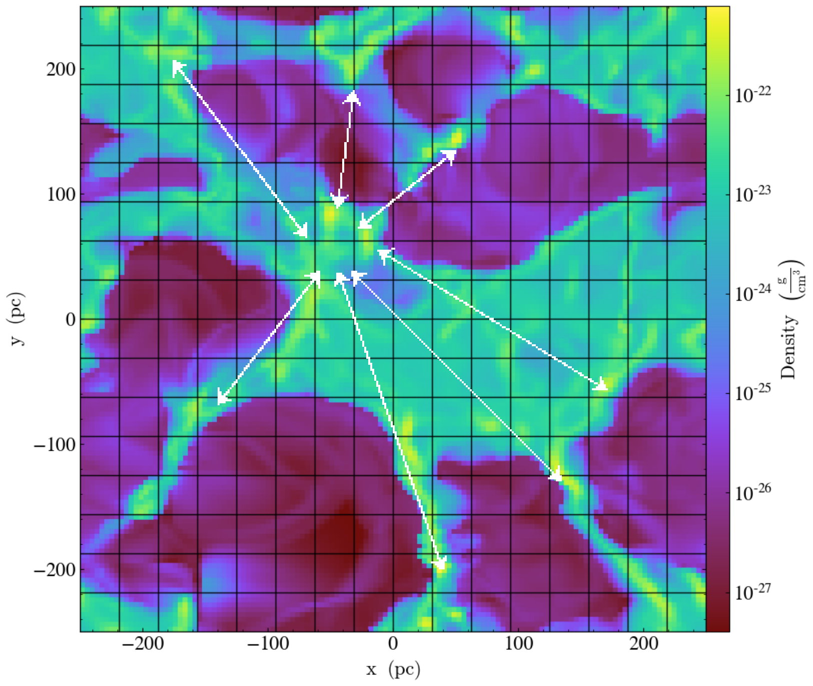


NN vs. long-range interaction

hydro (NN)
cell-by-cell speed



gravity (long-range)
instantaneous speed



direct neighbour vs. long-range

hydro (dir. neighbour)
cell-by-cell speed

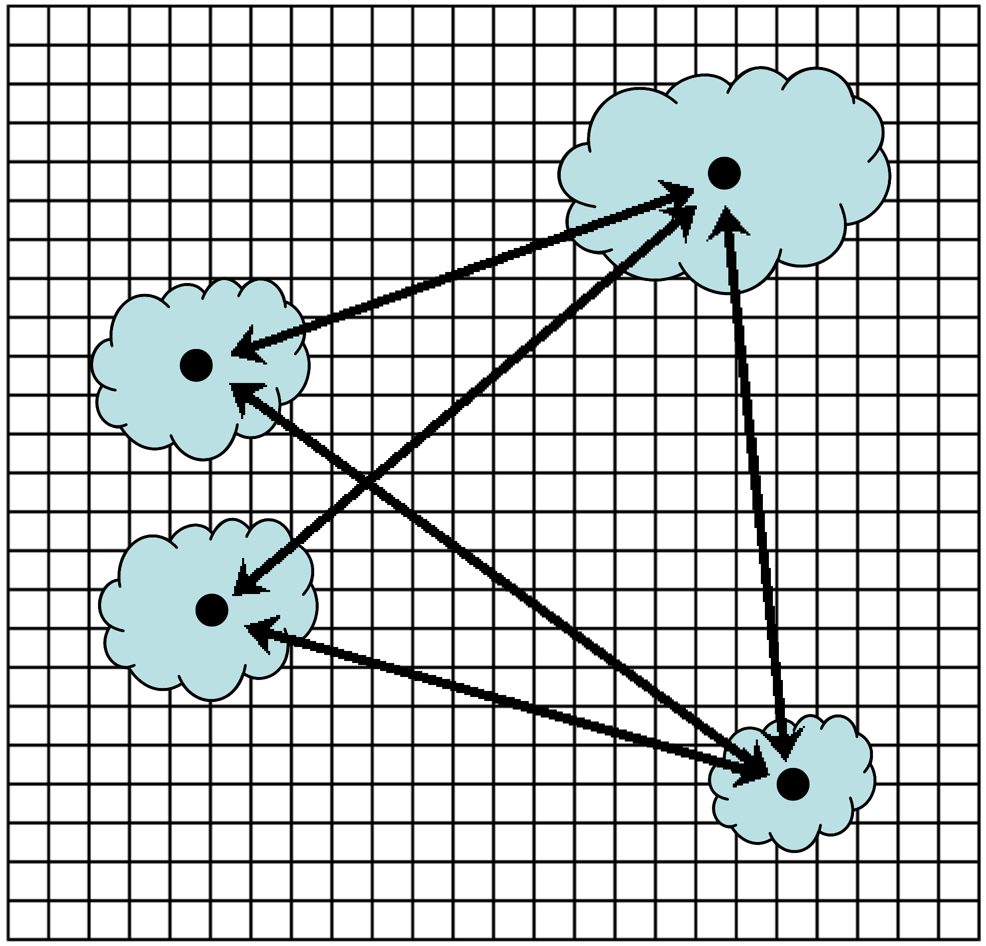
- one guard cell works
- two in case of 5pt stencil
- small additional memory
- communication to neighbour processors, globally asynchronous

gravity (long-range)
instantaneous speed

- every cell depends on every cell (N^2)
- every processor need entire grid information
- reduce information, approximate computation
- tree methods, particle-mesh methods
- still communication accross all processors

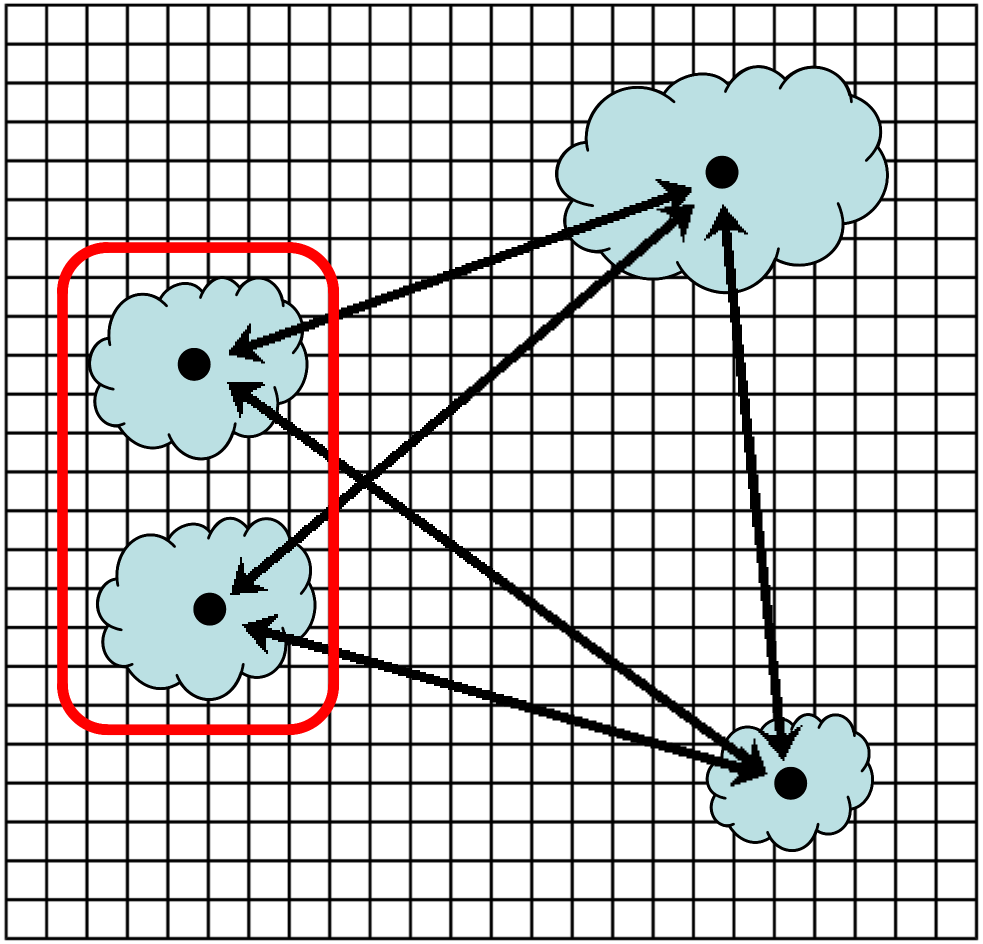
simple example: tree gravity

- reduce objects at large distances to centre of mass
- compute force between centres

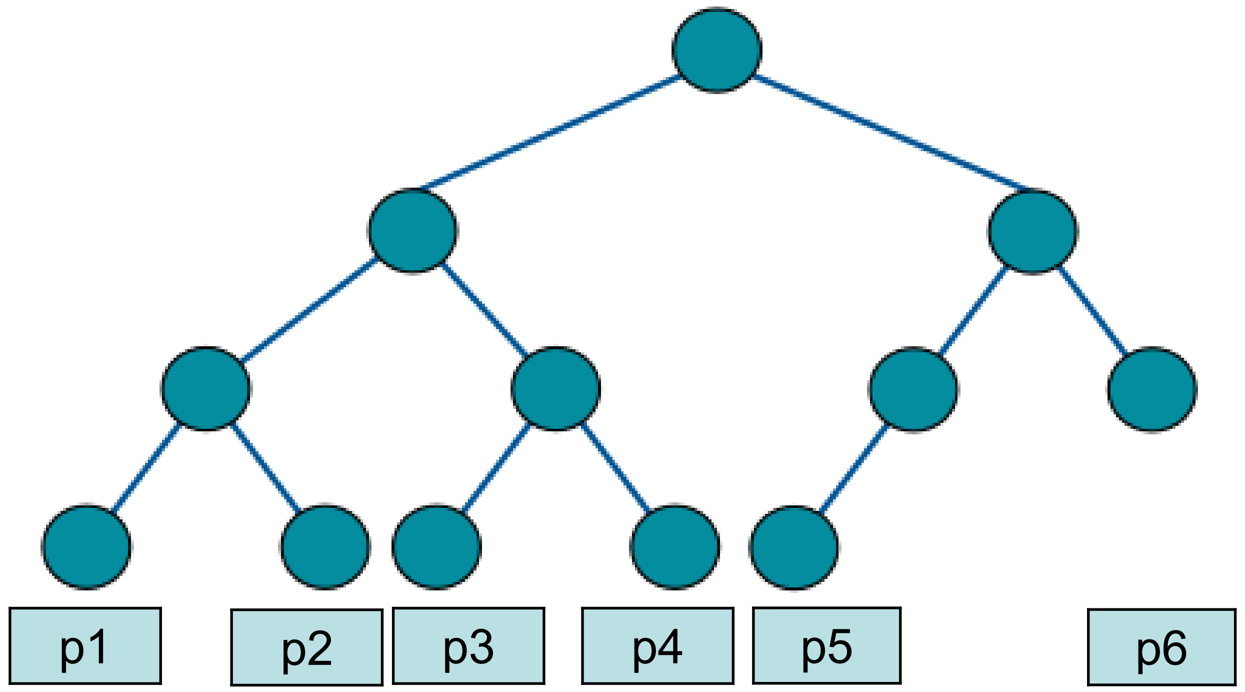


simple example: tree gravity

- reduce objects at large distances to centre of mass
- compute force between centres
- close clouds need direct integration



Tree communication



if tree structure is known:

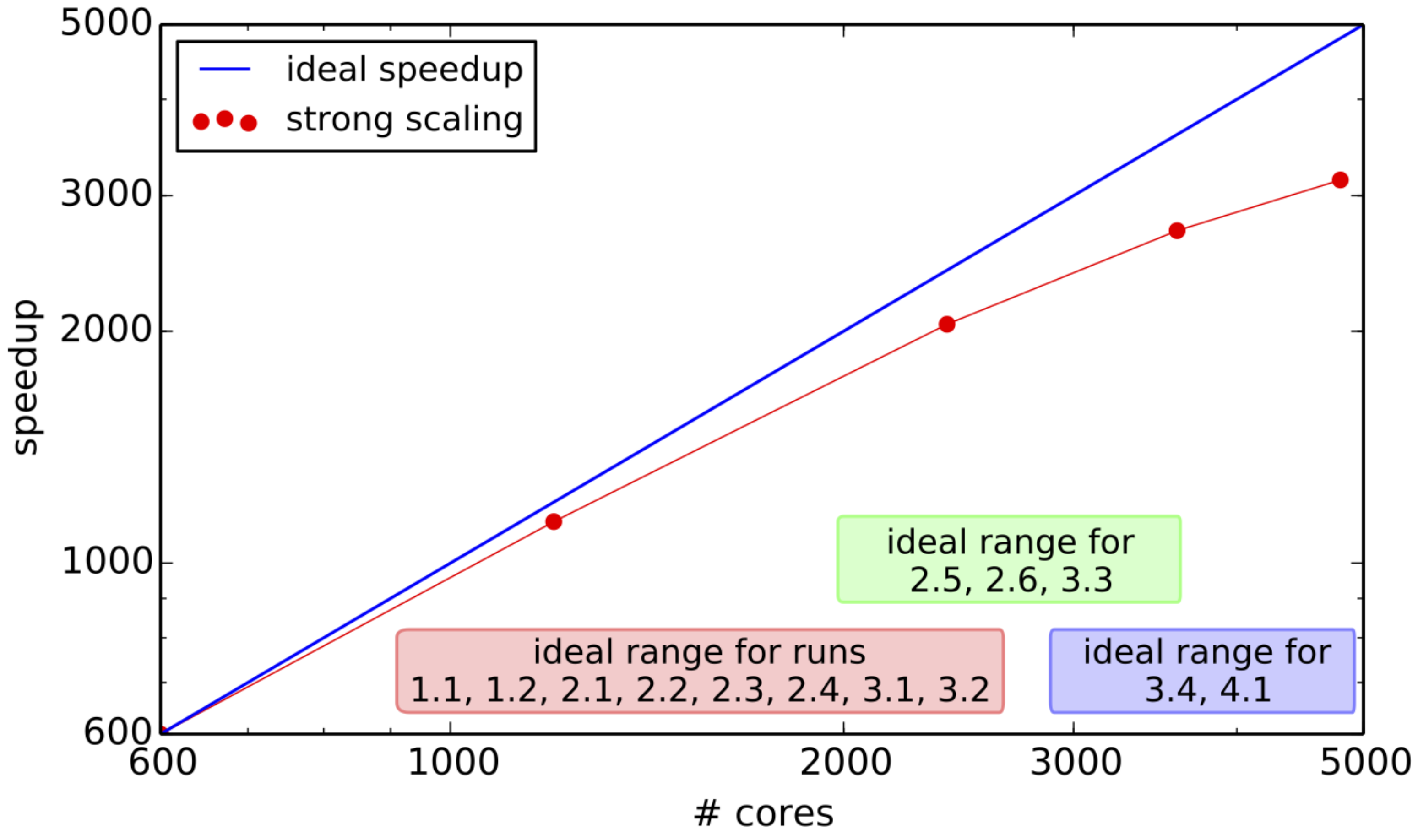
- pro: efficient communication with necessary processors
- con: every process needs to have tree information
  --> tree needs to be communicated

scaling

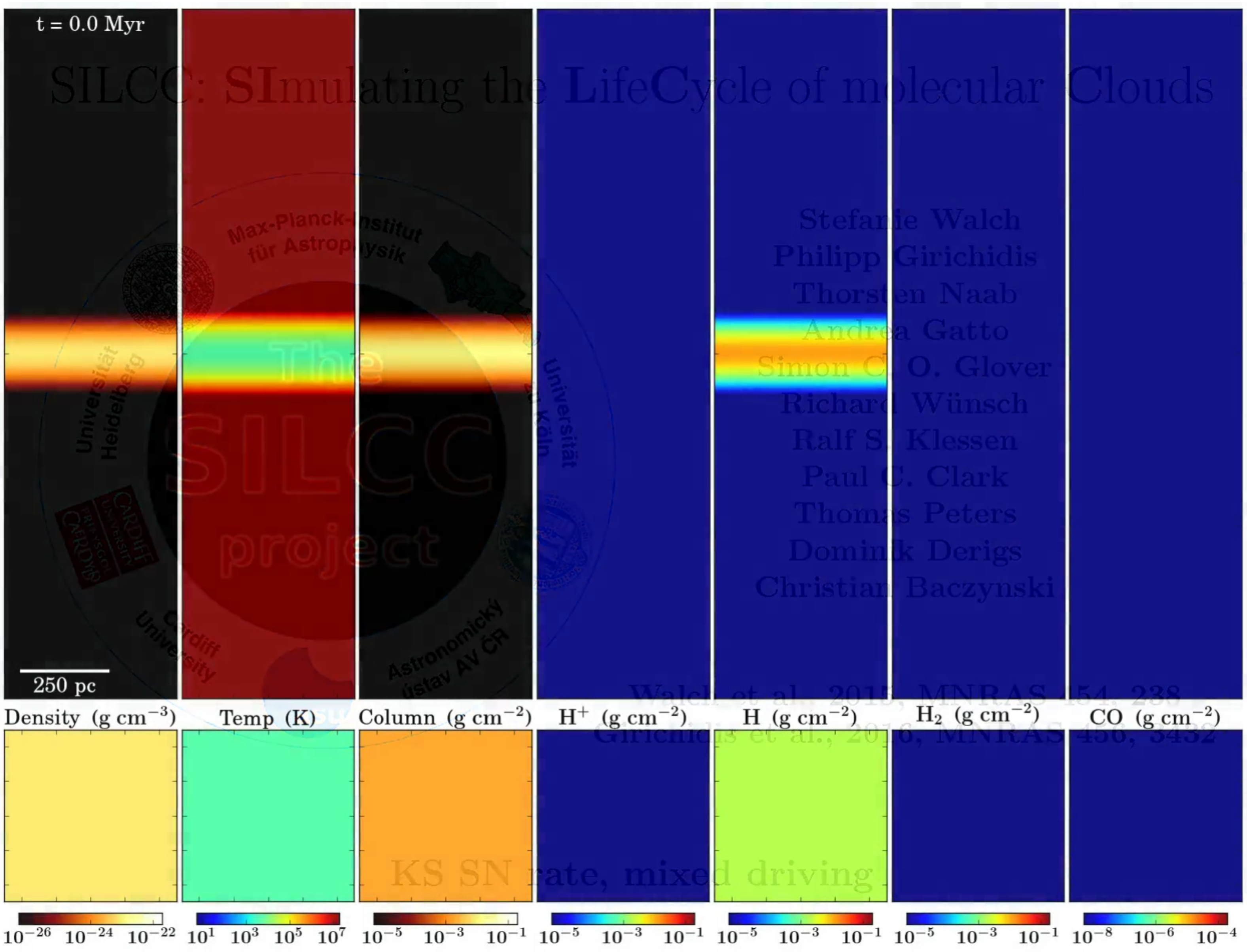
- *strong scaling*: how the solution time varies with the number of processors for a *fixed total problem size*
  - ideal: $t = N_{tot}/N_{proc}$, speedup = $1/t = N_{proc}/N_{tot}$

- *weak scaling*: how the solution time varies with the number of processors for a *fixed problem size per processor*
  - ideal: t = const, speedup = const

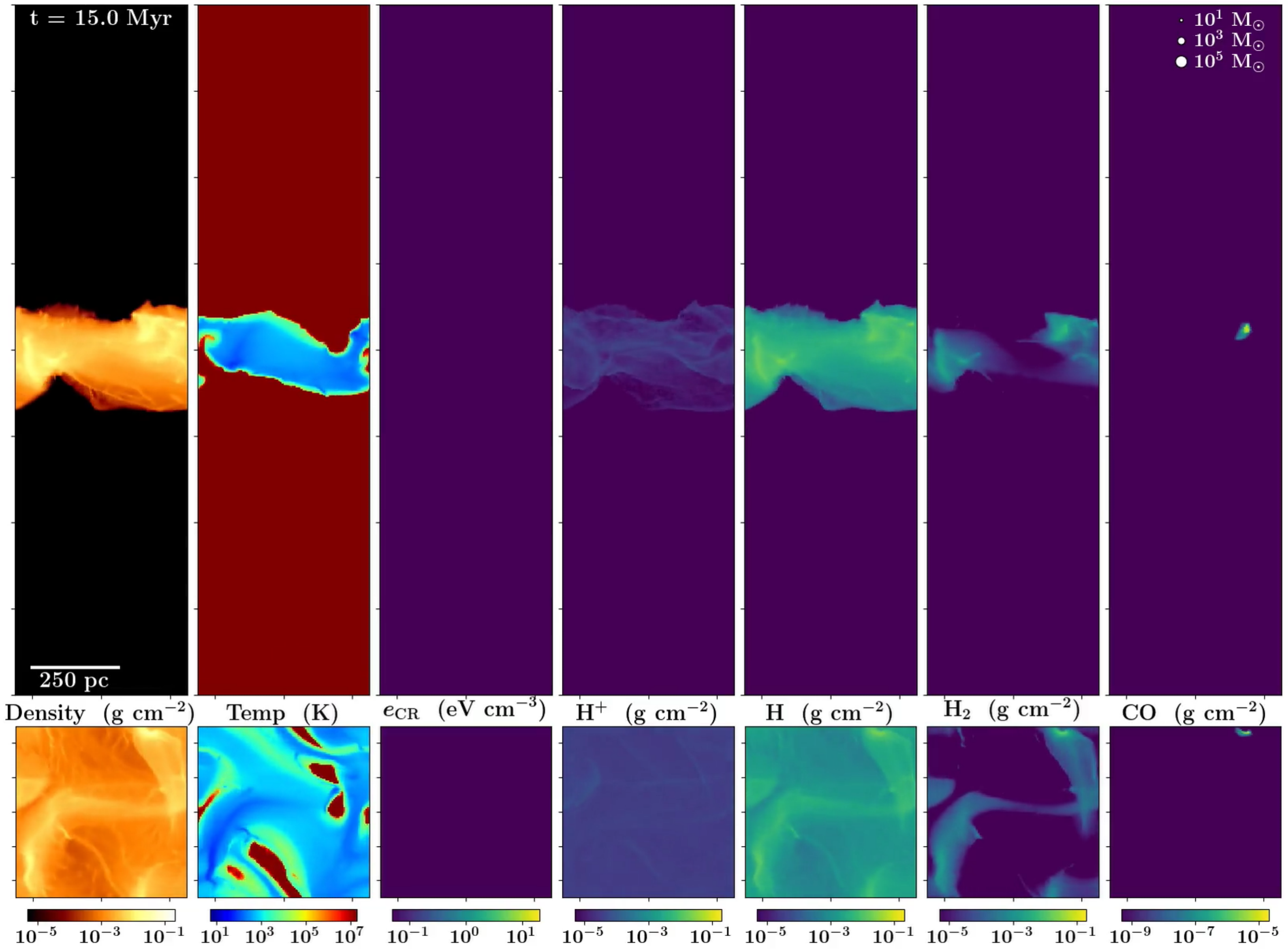
scaling in real application



ISM simulations



ISM simulations



processes

- MHD (local)
- self-gravity* (tree)
- external potential (analytic)
- radiation* and shielding* (tree)

- in practice:
  - tree efficient in terms of comp. cost
  - tree stores variables for (*), a lot of memory
  - sim. “memory limited”
  - more cores *would* help, but not enough memory

IO

- reading data:
  - few thousand cores direct reading efficient caches -> OK
  - one process reads -> MPI distribution
- writing data:
  - parallel writing at random positions in file: data race! (only one process allowed, lock)
  - files split like domain decomposition (every processor separate file with local data)
  - one process: MPI collection -> writing

IO

- reading data:
  - few thousand cores direct reading
    efficient caches -> OK
  - one process reads -> MPI distribution
- writing data:
  - parallel writing at random positions in file:
    data race! (only one process allowed, lock)
  - files split like domain decomposition
    (every processor separate file with local data)
  - one process: MPI collection -> writing

code

- FLASH / Arepo
- C / C++ / Fortran
- MPI / MPI+OpenMP
- ca. 400.000 lines
- problem: 100.000 lines
- current sim: 40 Mio CPUh, 250 TB

problems and conclusions

- all computations must be parallel
- many runs need >1000 cores
- MPI and combined MPI/openMP
- most of work:
  - numerical methods for the physics equations
  - optimization and efficient parallelization

- so far missing: machine learning methods