SESSION 3-4

Time evolution

Hybrid Kinetic Formalism (reminder)

Faraday's law

$$\frac{\partial \mathbf{B}}{\partial t} = -\nabla \times \mathbf{E},$$

Ampere's law

$$\mu_0 \mathbf{j} = \nabla \times \mathbf{B},$$

Ion Vlasov eq.

$$\frac{\partial f_p}{\partial t} = -\mathbf{v} \cdot \nabla f_p - \frac{\mathbf{E} + \mathbf{v} \times \mathbf{B}}{m_p} \cdot \nabla_{\mathbf{v}} f_p,$$

Ion moments

$$n_i = \sum_p \int f_p(\mathbf{r}, \mathbf{v})\, d^3v,$$

$$\mathbf{v}_i = \frac{1}{n_i} \sum_p \int \mathbf{v}\, f_p(\mathbf{r}, \mathbf{v})\, d^3v,$$

Quasi-neutrality

$$n_i = n_e = n,$$

$$\mathbf{v}_e = \mathbf{v}_i - \frac{\mathbf{j}}{ne},$$

Generalized Ohm's law

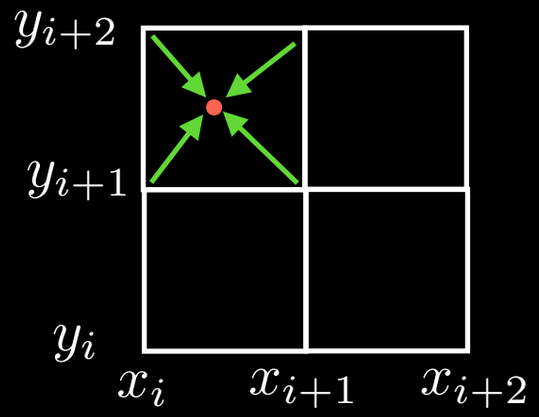
$$\mathbf{E} = -\mathbf{v}_e \times \mathbf{B} - \frac{1}{en} \nabla \cdot \mathbf{P}_e - \frac{m_e}{e} \frac{d\mathbf{v}_e}{dt}.$$

Move particles

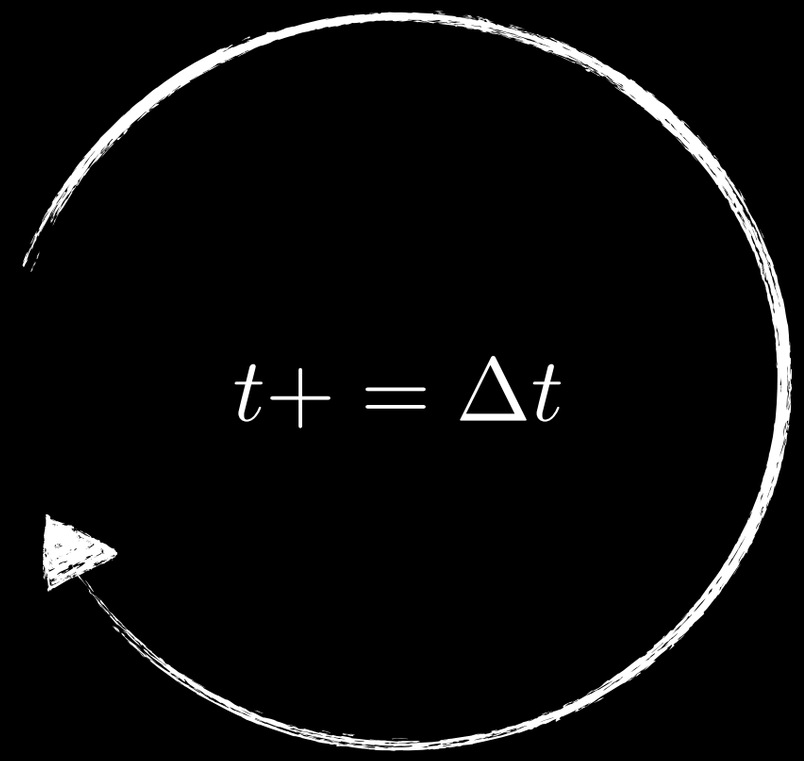
$$m\frac{d\mathbf{v}}{dt} = q\left(\mathbf{E} + \mathbf{v_p} \times \mathbf{B}\right) \qquad \frac{d\mathbf{r_p}}{dt} = \mathbf{v_p}$$

Fields to particles

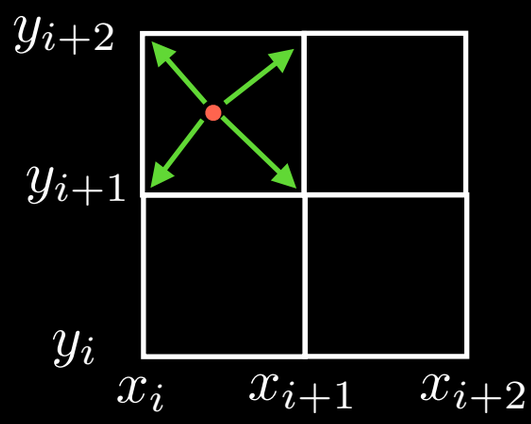
$(\mathbf{E}(\mathbf{r}), \mathbf{B}(\mathbf{r}))$



$$\mathbf{Q}(\mathbf{r}) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} Q_{ij}\, S(\mathbf{r_p} - \mathbf{r_{ij}})$$



Accumulate moments



$$n_{ij} = \sum_p^N S\left(\mathbf{r_p} - \mathbf{r_{ij}}\right) w_p$$

$$\mathbf{v}_{ij} = \frac{1}{n_{ij}} \sum_p^N \mathbf{v_p} S\left(\mathbf{r_p} - \mathbf{r_{ij}}\right) w_p$$

Field equations

$$\frac{\partial \mathbf{B}}{\partial t} = -\nabla \times \mathbf{E}, \qquad \mathbf{v}_e = \mathbf{v}_i - \frac{\mathbf{j}}{ne},$$

$$\mu_0\, \mathbf{j} = \nabla \times \mathbf{B}, \qquad \mathbf{E} = -\mathbf{v}_e \times \mathbf{B} - \frac{1}{en}\nabla \cdot \mathbf{P}_e - \frac{m_e}{e}\frac{d\mathbf{v}_e}{dt}.$$

The PIC loop

population.hpp

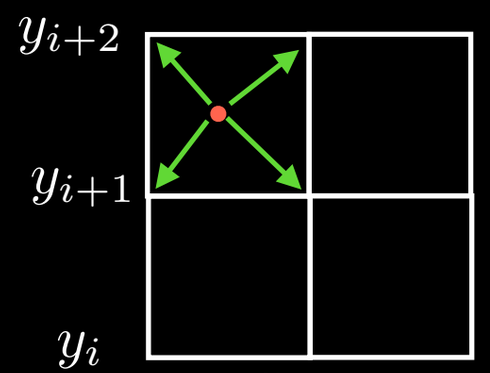
```
void deposit()
{
    static_assert(dimension == 1, "Population only implemented for 1D");
    for (auto& n : m_density)
    {
        n = 0.0; // Reset the field
    }

    for (auto& fx : m_flux.x)
        fx = 0.0;
    for (auto& fy : m_flux.y)
        fy = 0.0;
    for (auto& fz : m_flux.z)
        fz = 0.0;

    for (auto const& particle : m_particles)
    {
        double const iCell_float = particle.position[0] / m_grid->cell_size(Direction::X);
        int const iCell_         = static_cast<int>(iCell_float);
        double const reminder    = iCell_float - iCell_;
        auto const iCell         = iCell_ + m_grid->dual_dom_start(Direction::X);


        // TODO implement linear weighting deposit for the density and flux
    }
}
```

Accumulate moments



$$n_{ij} = \sum_p^N S\left(\mathbf{r_p} - \mathbf{r_{ij}}\right) w_p$$

$$\mathbf{v}_{ij} = \frac{1}{n_{ij}} \sum_p^N \mathbf{v_p} S\left(\mathbf{r_p} - \mathbf{r_{ij}}\right) w_p$$

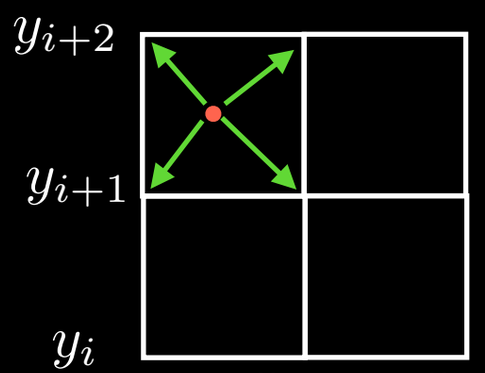
The PIC loop

moments.hpp

```
template<std::size_t dimension>
void total_density(std::vector<Population<dimension>> const& populations, Field<dimension>& N)
{
    for (auto ix = 0; ix < N.data().size(); ++ix)
    {
        N(ix) = 0;
    }
    for (auto const& pop : populations)
    {
        // TODO calculate the total density
    }
}

template<std::size_t dimension>
void bulk_velocity(std::vector<Population<dimension>> const& populations, Field<dimension> const& N,
                   VecField<dimension>& V)
{
    for (auto ix = 0; ix < N.data().size(); ++ix)
    {
        V.x(ix) = 0;
        V.y(ix) = 0;
        V.z(ix) = 0;
    }
    for (auto& pop : populations)
    {
        for (auto ix = 0; ix < N.data().size(); ++ix)
        {
            V.x(ix) += pop.flux().x(ix);
            V.y(ix) += pop.flux().y(ix);
            V.z(ix) += pop.flux().z(ix);
        }
    }
    // TODO calculate bulk velocity by dividing by density N
}
```

Accumulate moments



$$n_{ij} = \sum_p^N S\left(\mathbf{r_p} - \mathbf{r_{ij}}\right) w_p$$

$$\mathbf{v}_{ij} = \frac{1}{n_{ij}} \sum_p^N \mathbf{v_p} S\left(\mathbf{r_p} - \mathbf{r_{ij}}\right) w_p$$

Iterated Crank Nicholson

$$\frac{\partial u}{\partial t} = \frac{\partial u}{\partial x}$$

$$\frac{u_j^{p_1,n+1} - u_j^n}{\Delta t} = \frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x}$$

$$u_j^{n+1/2} = \frac{1}{2}\left(u_j^{p_1,n+1} + u_j^n\right)$$

$$\frac{u_j^{p_2,n+1} - u_j^n}{\Delta t} = \frac{u_{j+1}^{p_1,n+1/2} - u_{j-1}^{p_1,n+1/2}}{2\Delta x}$$

$$u_j^{n+1/2} = \frac{1}{2}\left(u_j^{p_2,n+1} + u_j^n\right)$$

$$\frac{u_j^{,n+1} - u_j^n}{\Delta t} = \frac{u_{j+1}^{p_2,n+1/2} - u_{j-1}^{p_2,n+1/2}}{2\Delta x}$$

https://arxiv.org/pdf/gr-qc/9909026

3 STEPS ITERATED CRANK NICHOLSON

t

Prediction

$$\mathbf{B}_{p1}^{n+1} = \mathbf{B}^n - \Delta t \nabla \times \mathbf{E}^n$$

$$\mathbf{E}_{p1}^{n+1} = -\mathbf{u}^n \times \mathbf{B}_{p1}^{n+1} + \frac{\nabla \times \mathbf{B}_{p1}^{n+1}}{N^n} - \frac{\nabla \cdot \mathbf{P_e}}{N^n} + \eta \nabla \times \mathbf{B}_{p1}^{n+1} - \nu \nabla^2 \nabla \times \mathbf{B_{p1}^{n+1}}$$

$$(\mathbf{E}, \mathbf{B})^{n+1/2} =< (\mathbf{E}, \mathbf{B}) >_n^{n+1}$$

$$\mathbf{r}_{p1}^{n+1/2} = \mathbf{r}^n + \Delta t/2 \mathbf{v}^n$$

$$\mathbf{E}, \mathbf{B}\left(\mathbf{r_{p1}^{n+1/2}}\right) = \sum_{ijk} \left(\mathbf{E}, \mathbf{B}_{ijk}\right) W\left(|\mathbf{r}_{ijk} - \mathbf{r}_{p1}^{n+1/2}|\right)$$

$$m_i \frac{d\mathbf{v}_{p1}^{n+1}}{dt} = e\left(\mathbf{v}^n \times + \mathbf{B}^{n+1/2} + \mathbf{E}^{n+1/2}\right)$$

$$N^{n+1} = \sum_p w_p \mathbf{v}_{p1}^{n+1} W\left(|\mathbf{r}_{ijk} - \mathbf{r}_{p1}^{n+1}|\right) \qquad u^{n+1} = \sum_p w_p \mathbf{v}_{p1}^{n+1} W\left(|\mathbf{r}_{ijk} - \mathbf{r}_{p1}^{n+1}|\right)$$

Prediction

$$\mathbf{B}_{p2}^{n+1} = \mathbf{B}^n - \Delta t \nabla \times \mathbf{E}^{n+1/2}$$

$$\mathbf{E}_{p2}^{n+1} = -\mathbf{u}^{n+1} \times \mathbf{B}_{p2}^{n+1} + \frac{\nabla \times \mathbf{B}_{p2}^{n+1}}{N^{n+1}} - \frac{\nabla \cdot \mathbf{P_e}}{N^{n+1}} + \eta \nabla \times \mathbf{B}_{p2}^{n+1} - \nu \nabla^2 \nabla \times \mathbf{B_{p2}^{n+1}}$$

$$\mathbf{r}_{p2}^{n+1/2} = \mathbf{r}^n + \Delta t/2 \mathbf{v}^n$$

$$(\mathbf{E}, \mathbf{B})^{n+1/2} =< (\mathbf{E}, \mathbf{B}) >_n^{n+1}$$

$$m_i \frac{d\mathbf{v}_{p2}^{n+1}}{dt} = e\left(\mathbf{v}^n \times + \mathbf{B}^{n+1/2} + \mathbf{E}^{n+1/2}\right)$$

$$N^{n+1} = \sum_p w_p \mathbf{v}_{p1}^{n+1} W\left(|\mathbf{r}_{ijk} - \mathbf{r}_{p1}^{n+1}|\right) \qquad u^{n+1} = \sum_p w_p \mathbf{v}_{p1}^{n+1} W\left(|\mathbf{r}_{ijk} - \mathbf{r}_{p1}^{n+1}|\right)$$

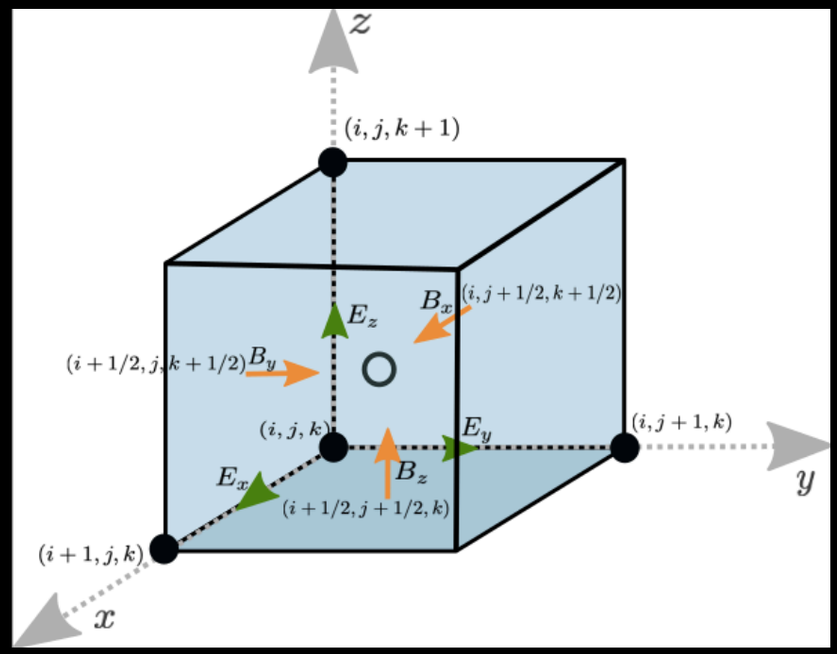
Correction

$$\mathbf{B}^{n+1} = \mathbf{B}^n - \Delta t \nabla \times \mathbf{E}^{n+1/2}$$

$$\mathbf{E}^{n+1} = -\mathbf{u}^{n+1} \times \mathbf{B}^{n+1} + \frac{\nabla \times \mathbf{B}^{n+1}}{N^{n+1}} - \frac{\nabla \cdot \mathbf{P_e}}{N^{n+1}} + \eta \nabla \times \mathbf{B}^{n+1} - \nu \nabla^2 \nabla \times \mathbf{B^{n+1}}$$

$t + \Delta t$

Yee grid

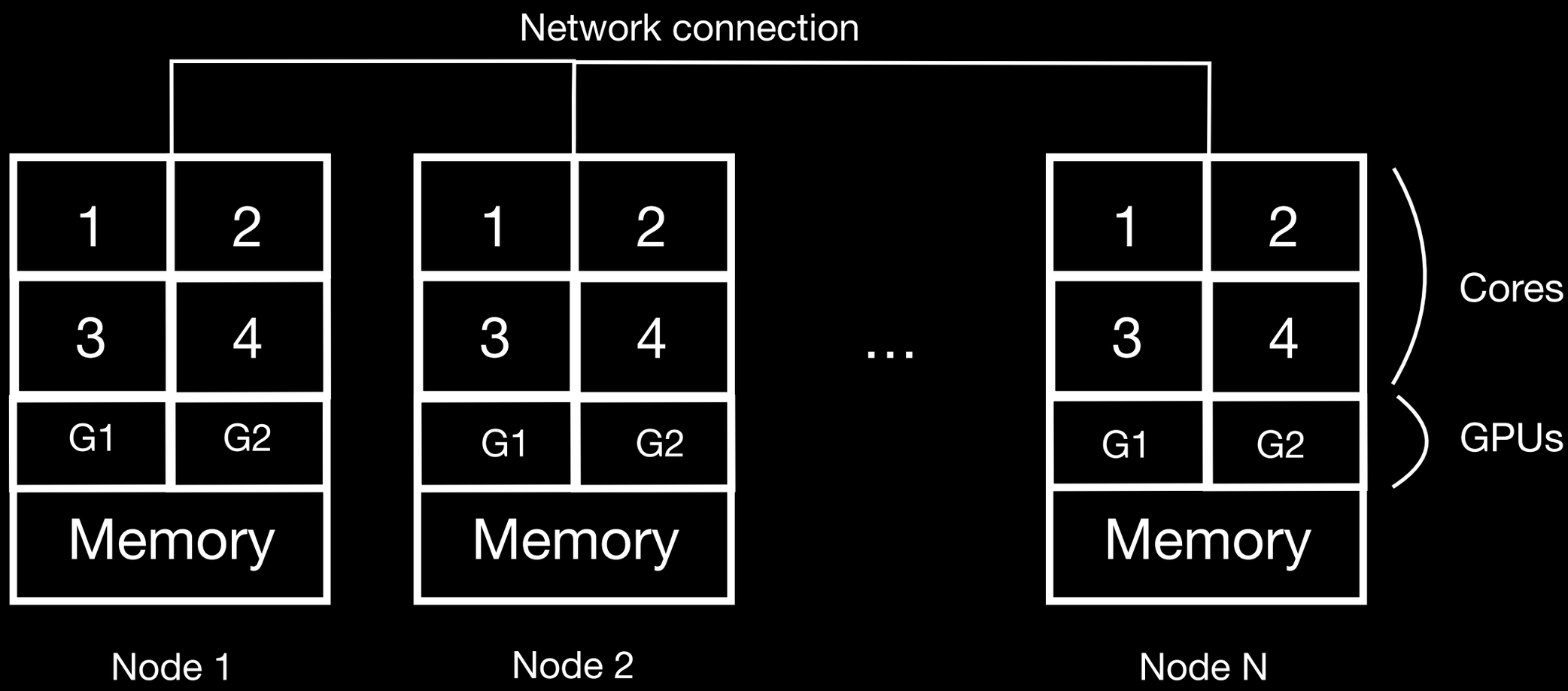


SESSION 4-5

Parallelism

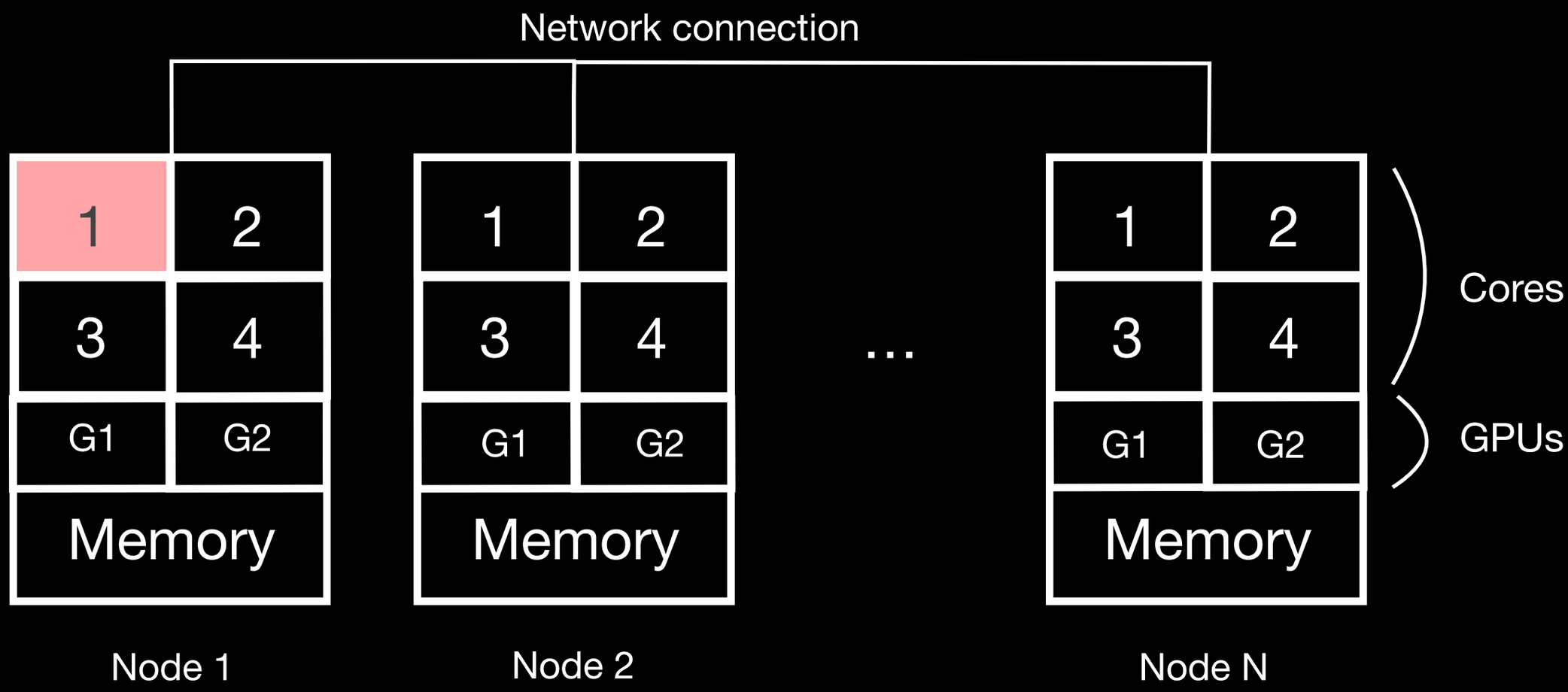
MPI and particle parallelization

There are many forms of parallelism



- Vectorization
- CPU threads
- Inter-node messaging

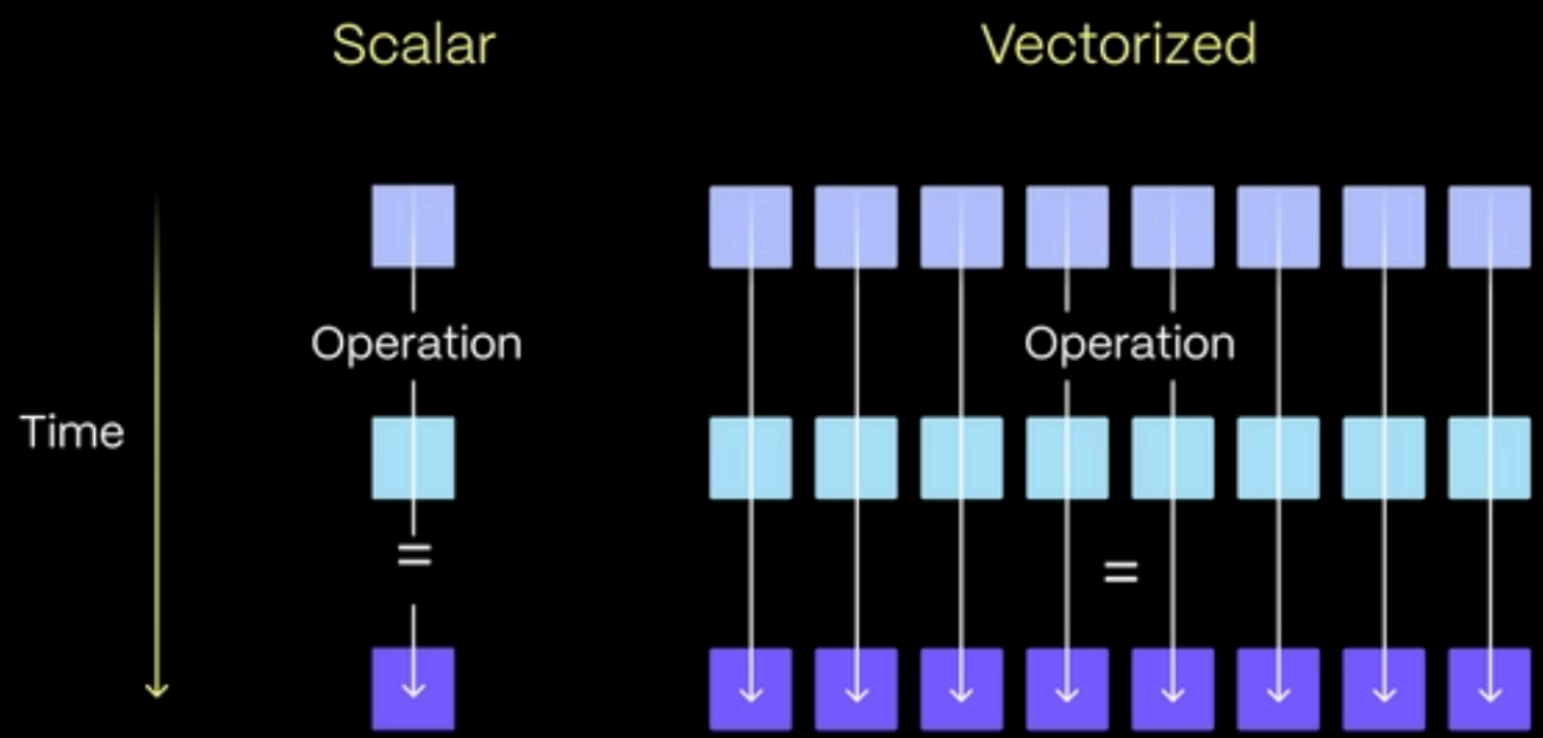
VECTORIZATION



- Vectorization
- CPU threads
- Inter-node messaging

VECTORIZATION

CPUs can actually do multiple operations at once in a vector fashion



Single Instruction Multiple Data : SIMD

VECTORIZATION

- your data structure needs to be « compatible » with vector operations (structures of arrays)
- Compilers can automatically vectorize your code

```
for (int i=0; i<16; ++i)
    C[i] = A[i] + B[i];
```

-O3 -march=native -mtune=native -fopt-info-vec

Manual (explicit) vectorization:

- Architecture dependent
- AVX512 : 8 double precision

```
for (int i=0; i<16; i+=4) {
    C[i]   = A[i]   + B[i];
    C[i+1] = A[i+1] + B[i+1];
    C[i+2] = A[i+2] + B[i+2];
    C[i+3] = A[i+3] + B[i+3];
}
```

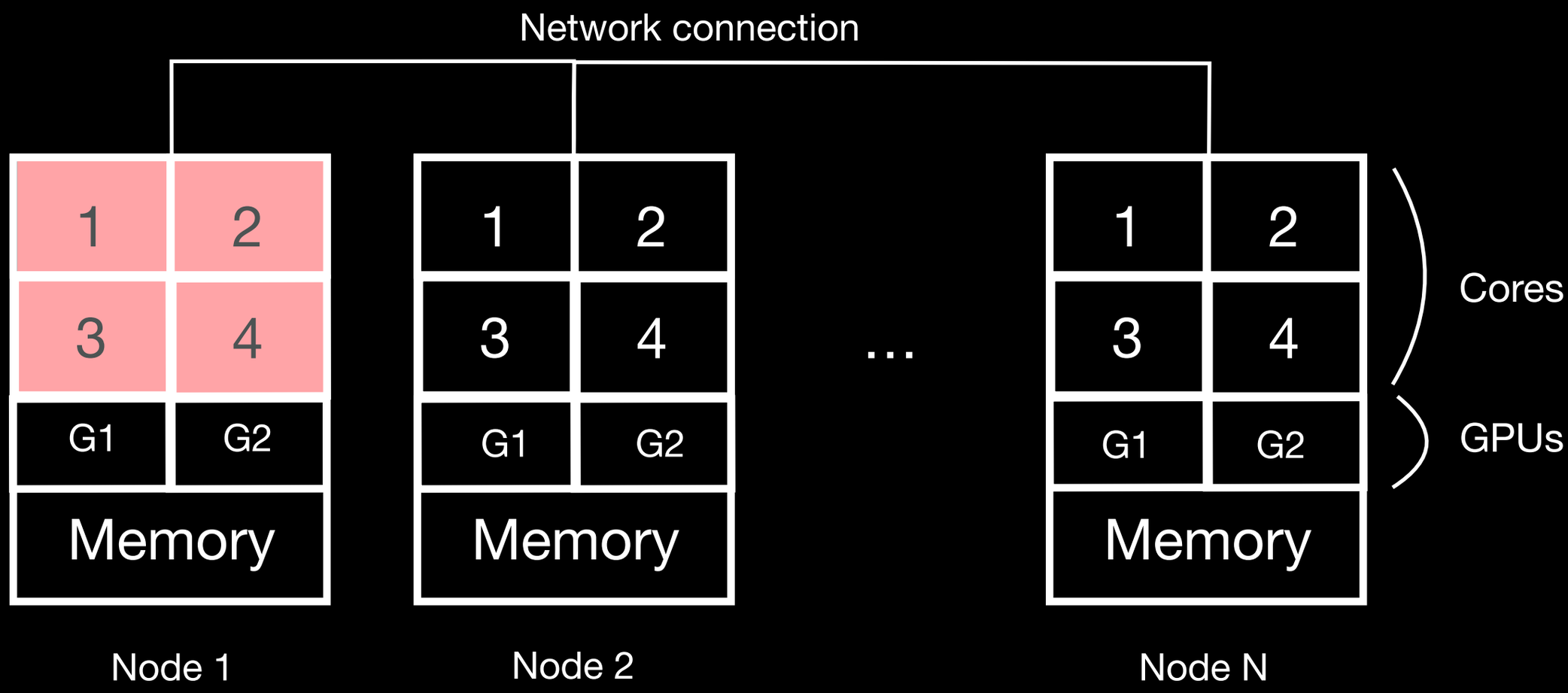
- Vectorization may speed up some parts of your code
- It's very low level, difficult
- Usually the last level of parallelism you care about

```
for (int i=0; i<16; i+=4)
    addFourThingsAtOnceAndStoreResult(&C[i], &A[i], &B[i]);
```

How to :

- Write vectorization intrinsics yourself (hard, not so portable)
- Use a portable SIMD libraries
- Use compiler directives (OpenMP, …)

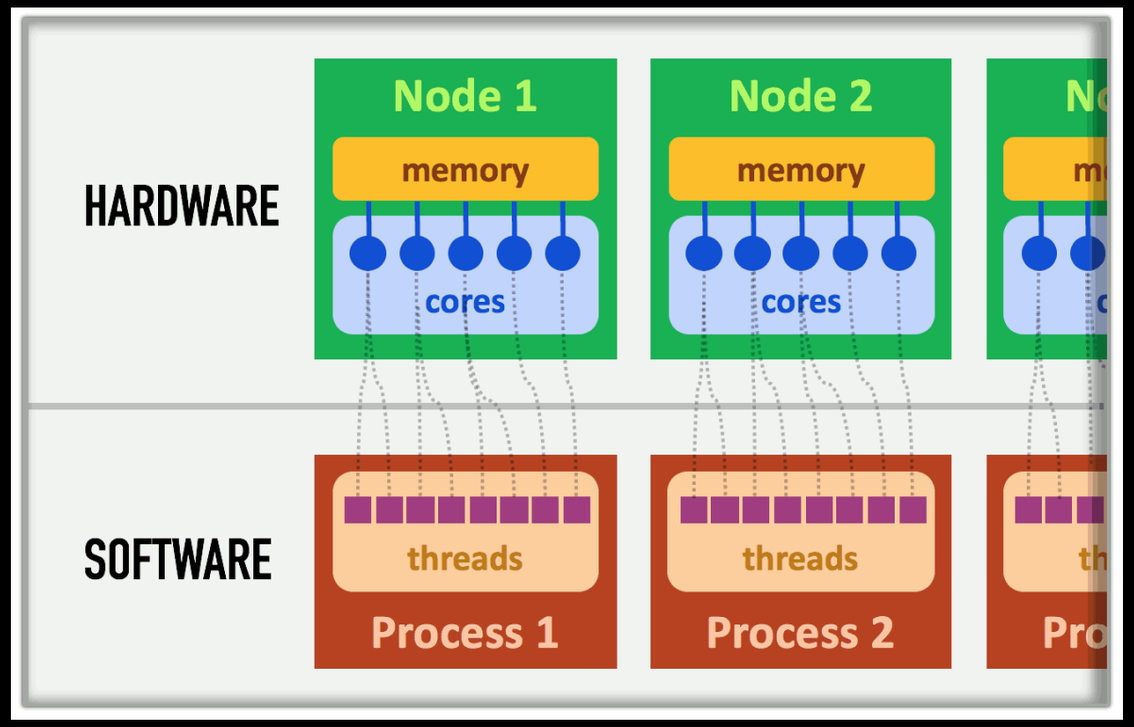
There are many forms of parallelism



- Vectorization
- CPU threads
- Inter-node messaging

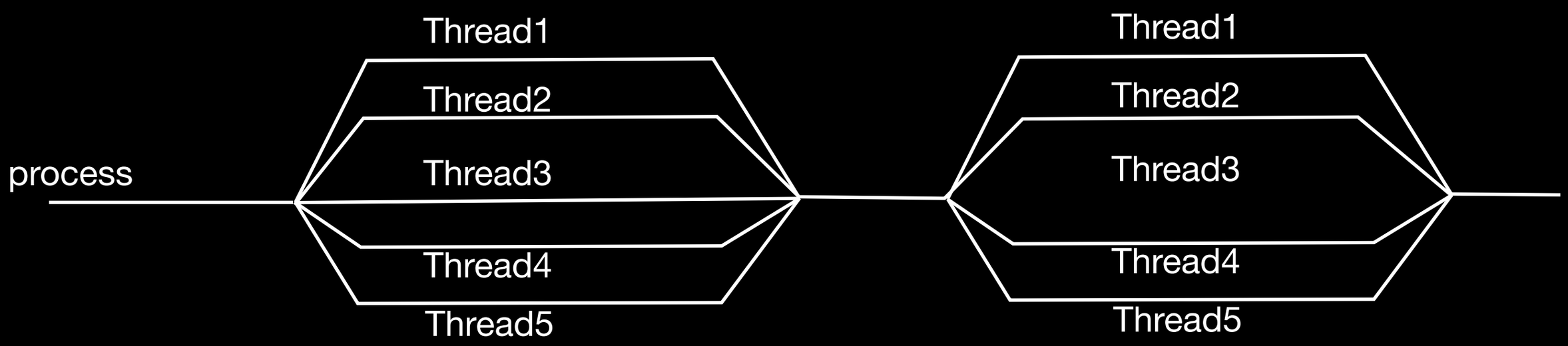
CPU MULTITHREADING (SHARED MEMORY PARALLELISM)

- Modern processors have **multiple cores**
- Cores **share the same memory**
- Cores can **do different things « simultaneously »**



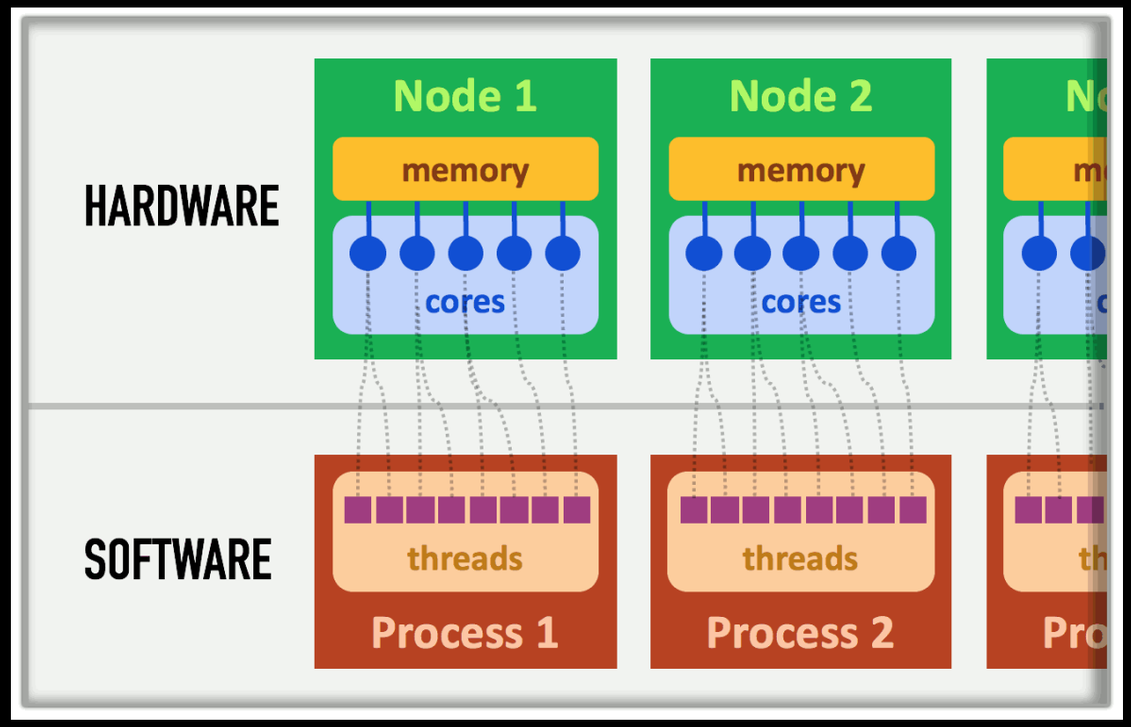
(From https://smileipic.github.io/Smilei/)

- A process can run **several *threads***
- A *thread* is a **software abstraction** to do parallelism
- You can have as many threads you want, typically you want **1 thread = 1 core**



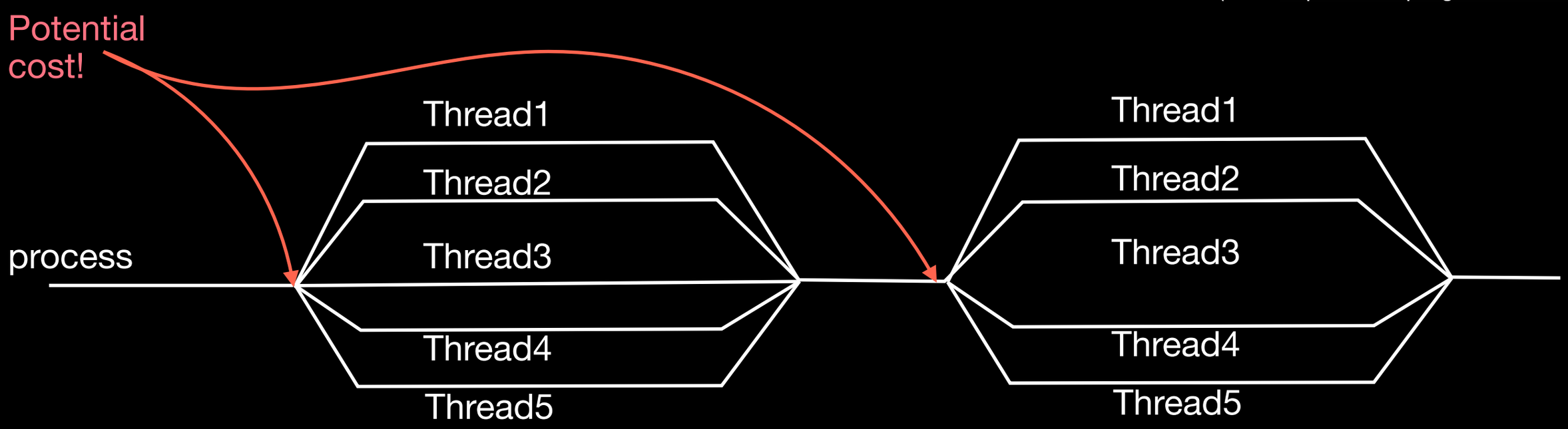
CPU MULTITHREADING (SHARED MEMORY PARALLELISM)

- Modern processors have **multiple cores**
- Cores **share the same memory**
- Cores can **do different things « simultaneously »**



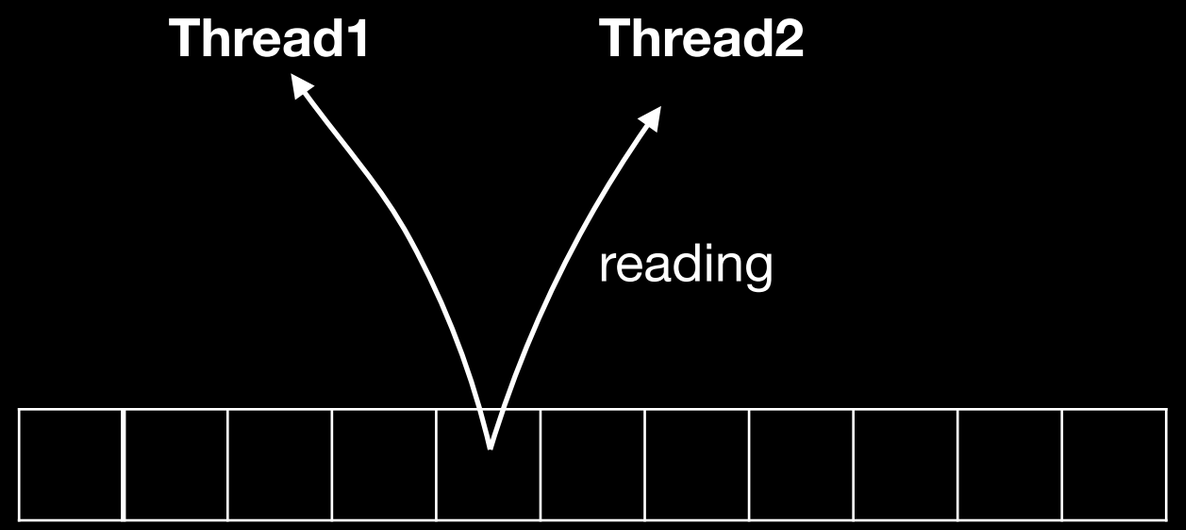
- A process can run **several *threads***
- A *thread* is a **software abstraction** to do parallelism
- You can have as many threads you want, typically you want **1 thread = 1 core**

(From https://smileipic.github.io/Smilei/)



Typical multithreading pitfalls: Race conditions

- Modern processors have **multiple cores**
- Cores **share the same memory**
- Cores can **do different things « simultaneously »**

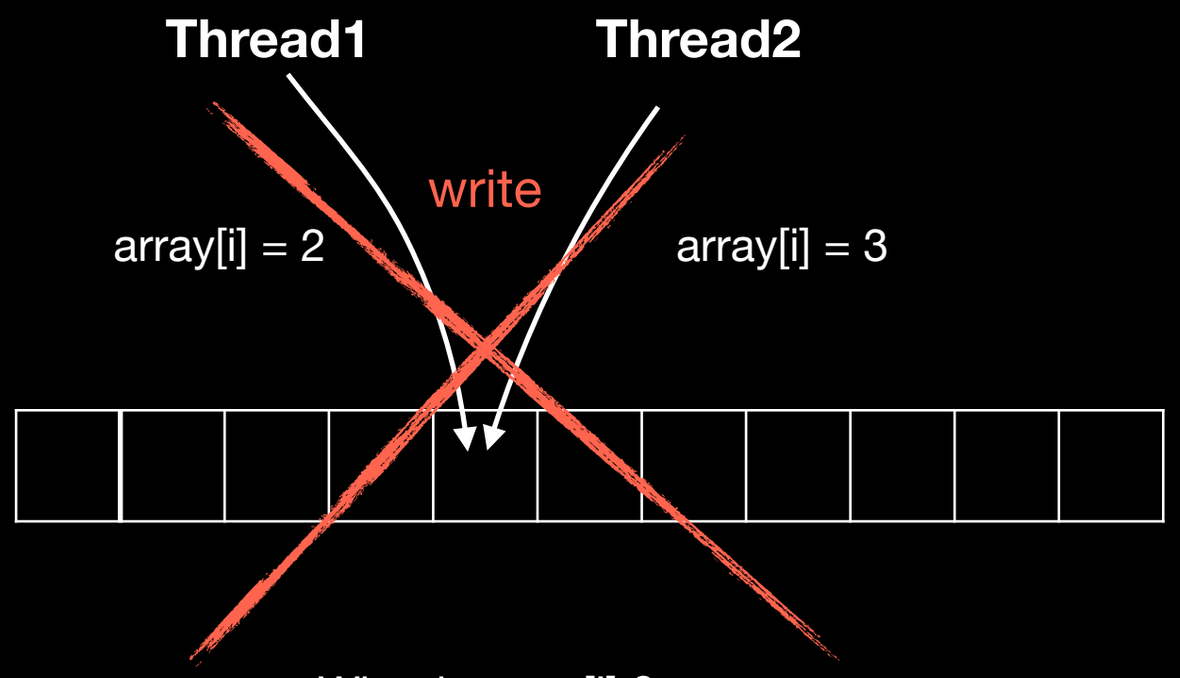


- A process can run **several *threads***
- A *thread* is a **software abstraction** to do parallelism
- You can have as many threads you want, typically you want **1 thread = 1 core**

Typical multithreading pitfalls: Race conditions

- Modern processors have **multiple cores**
- Cores **share the same memory**
- Cores can **do different things « simultaneously »**

- A process can run **several *threads***
- A *thread* is a **software abstraction** to do parallelism
- You can have as many threads you want, typically you want **1 thread = 1 core**



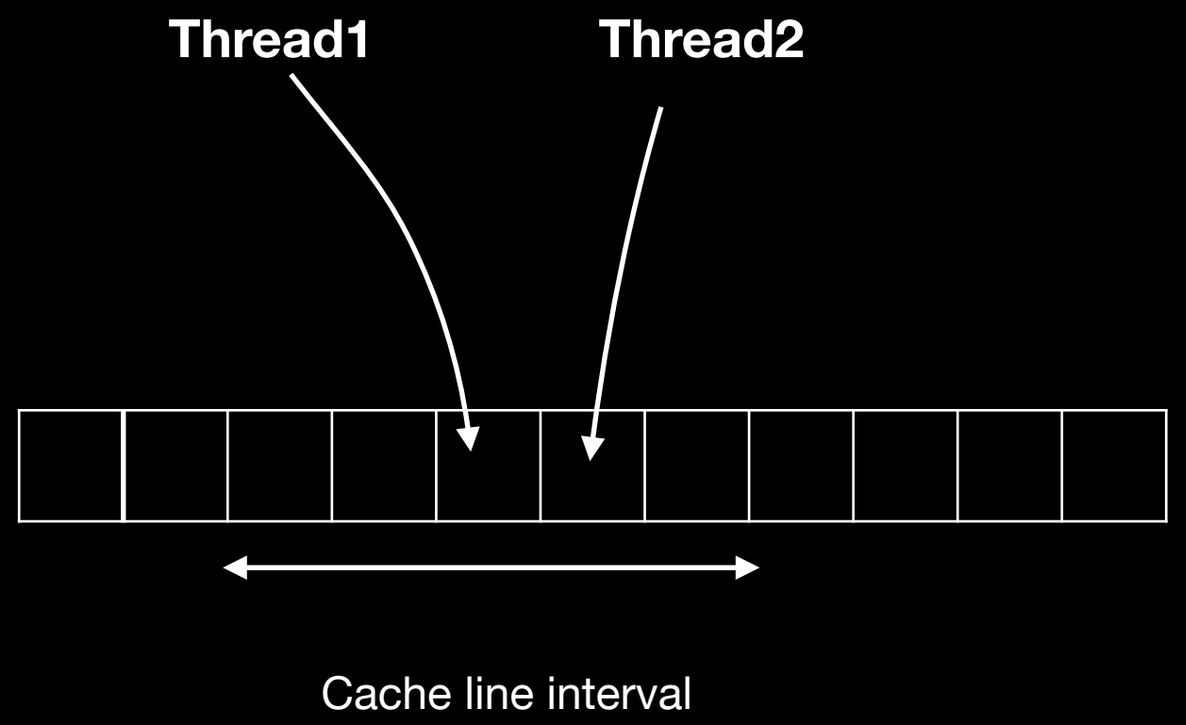
What is array[I] ?

« Race condition » (concurrent access)

—> « atomic » (sequential) access

Typical multithreading pitfalls: False Sharing

- Two theads access nearby elements that fall in the same cache line interval
- Cache is invalidated at each access
- Destroys performance
- Solution : cache alignment, padding, …



Other Typical multithreading pitfalls…

Order of operations

- (A+B) +C != A+(B+C)
- Order of operations done in parallel is NOT guaranteed
- Rounding errors will be different each time you run your program

Random numbers

- Random number are pseudo-random series
- If not using random seeds, series will be identical on different threads (or proc.), probably useless
- If using random seeds, debugging will be non-deterministic and very difficult

Debugging

- Is painful…
- Special debuggers : ddt, totalview… or prints with thread ID

CPU MULTITHREADING (SHARED MEMORY PARALLELISM)

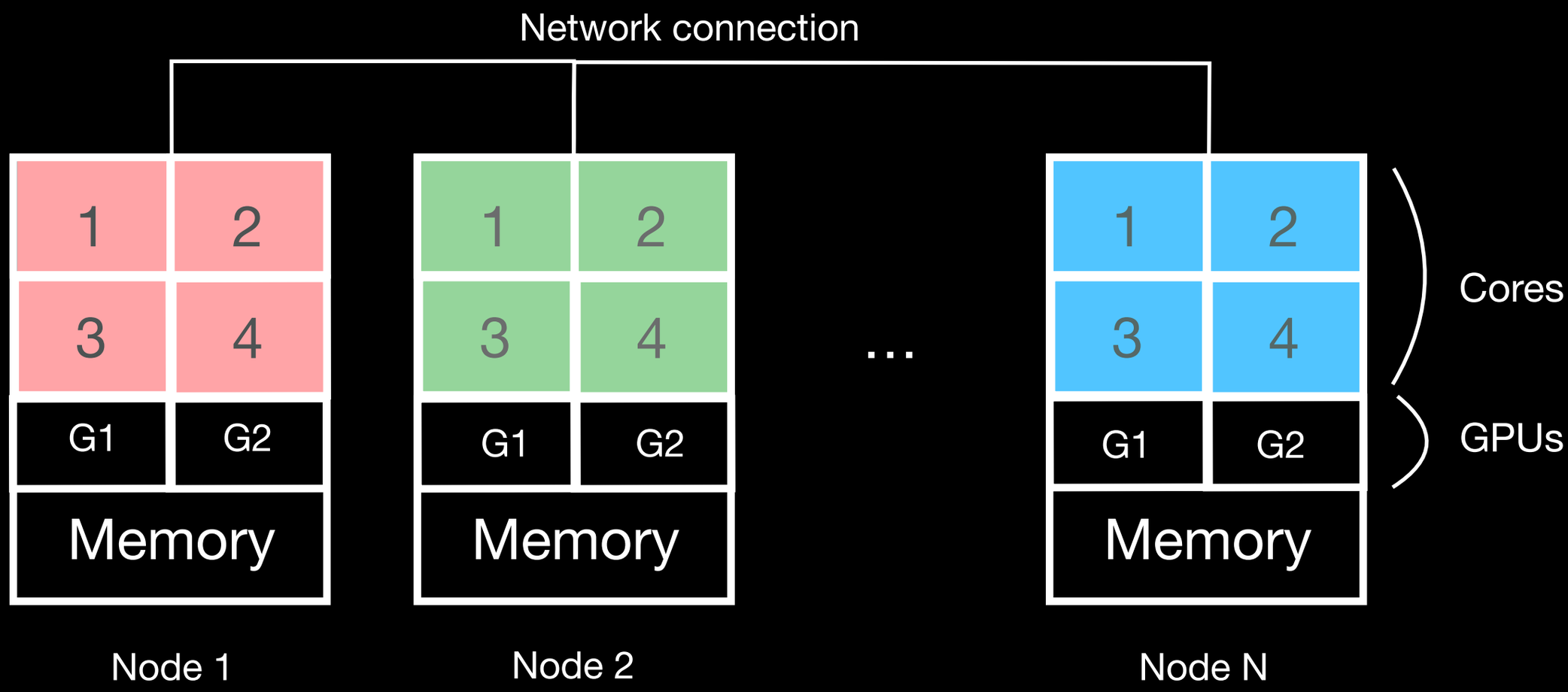
Different usage of threads

- **Data parallelism**
  - Push N particles on P threads
  - Update N magnetic field nodes on P threads…
- **Task parallelism**
  - Push particles while dumping diagnostics
  - …

How to?

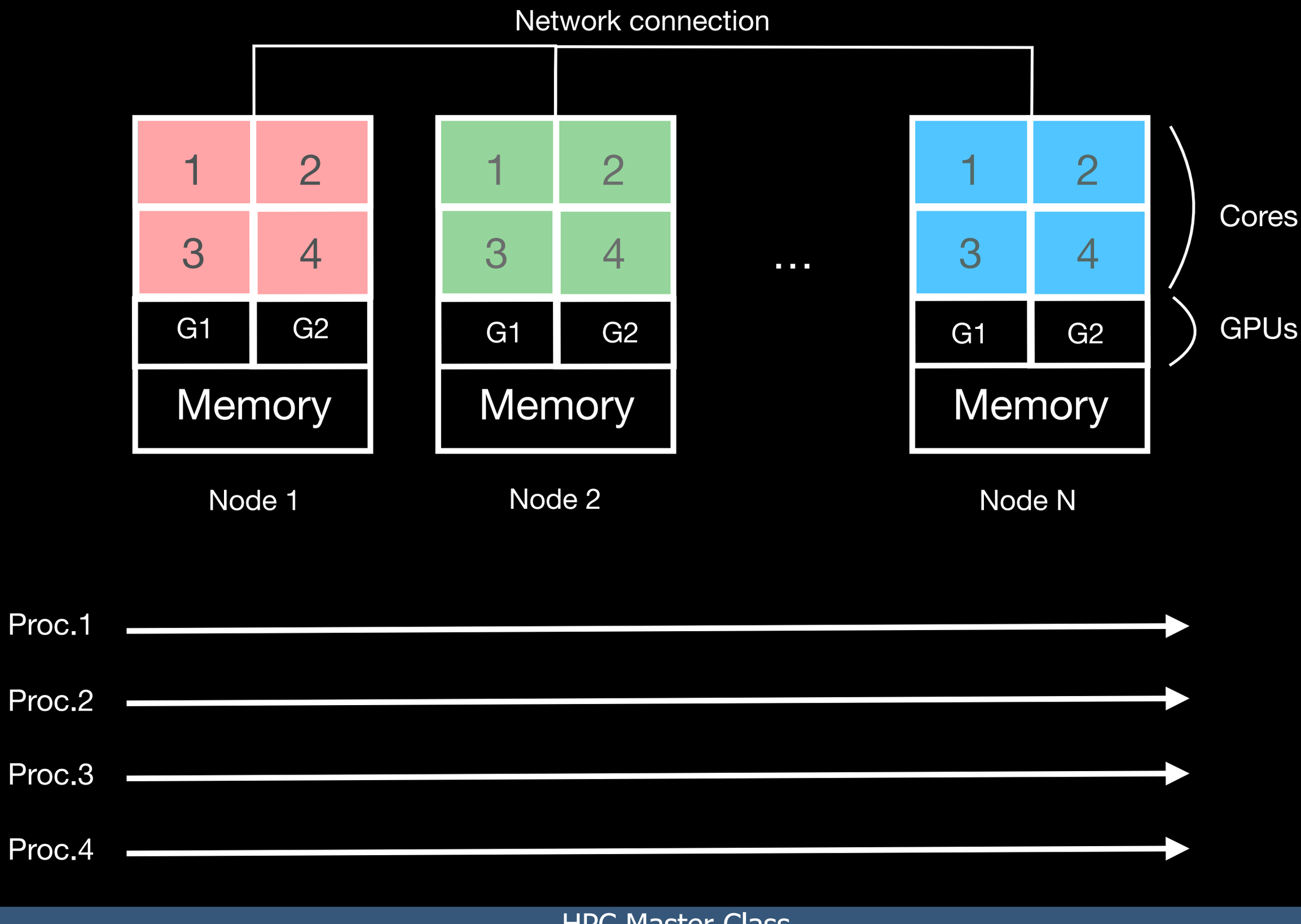
- **Compiler directives**: OpenMP, OpenACC
  - Describe parallelism of code blocks and the compiler does it for you
  - Simple to write, hard to optimize
- **Thread libraries**
  - Standard C++ threads, TBB, pthread
  - You have control, more difficult to write, easier to optimize

Distributed parallelism

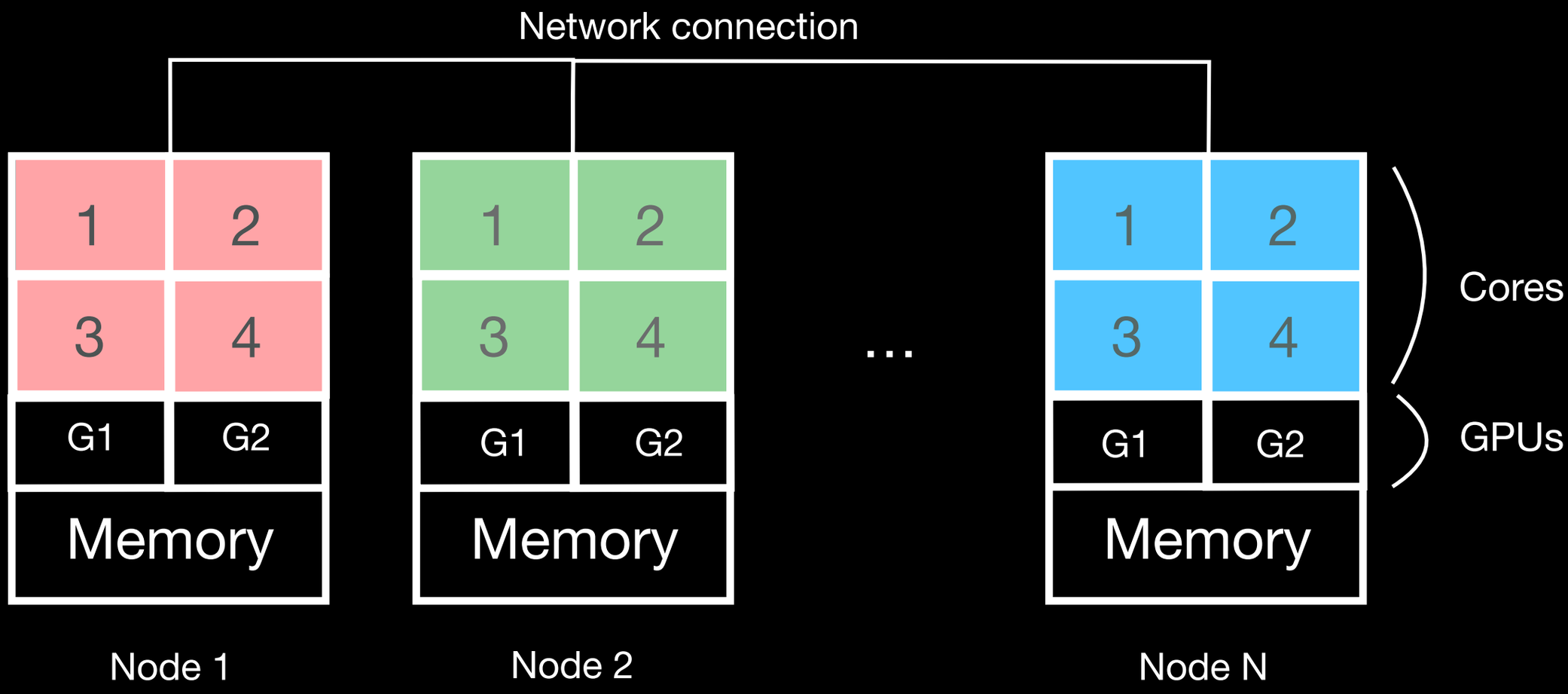


- Vectorization
- CPU threads
- Inter-node messaging

Distributed parallelism



Distributed parallelism



Embarrassingly parallel

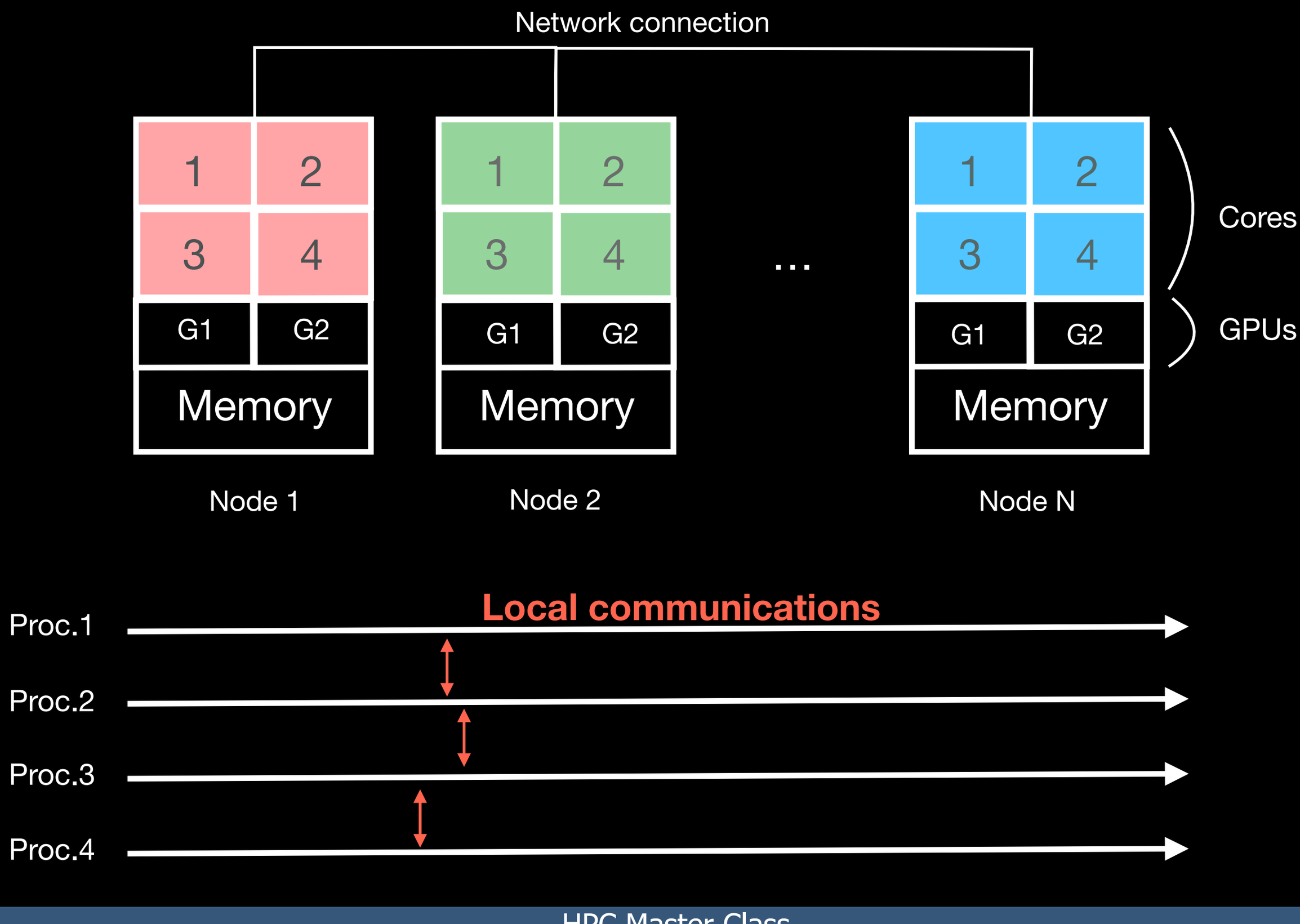
Proc.1

Proc.2

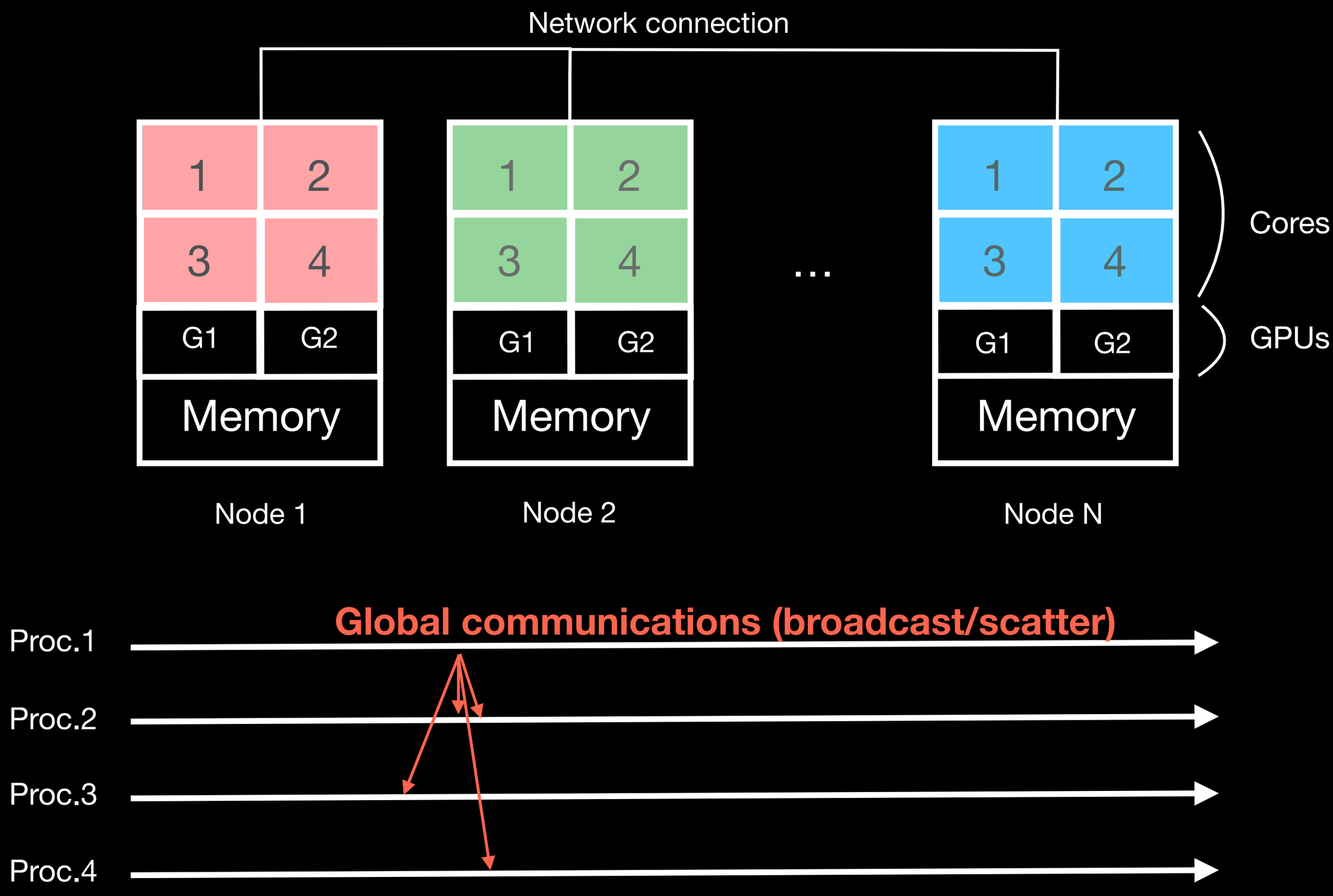
Proc.3

Proc.4

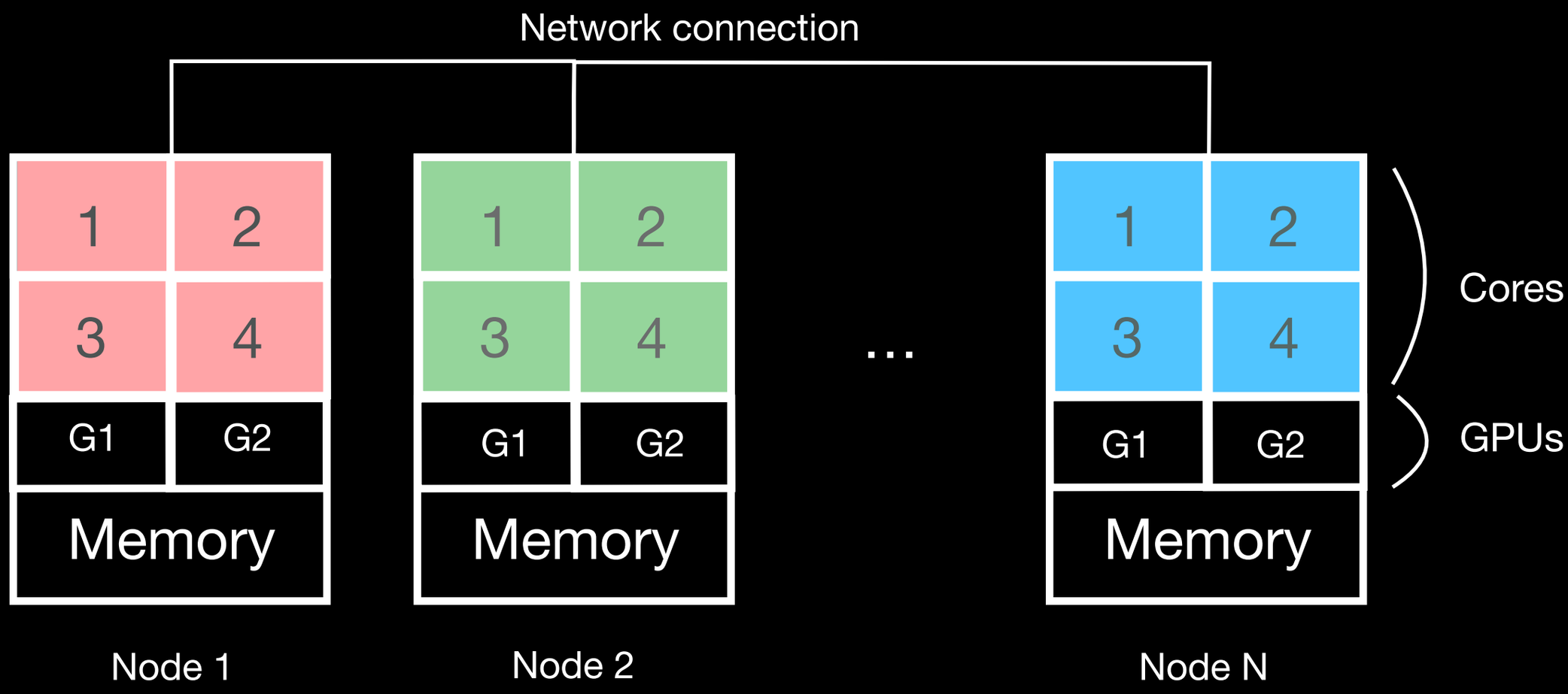
Distributed parallelism

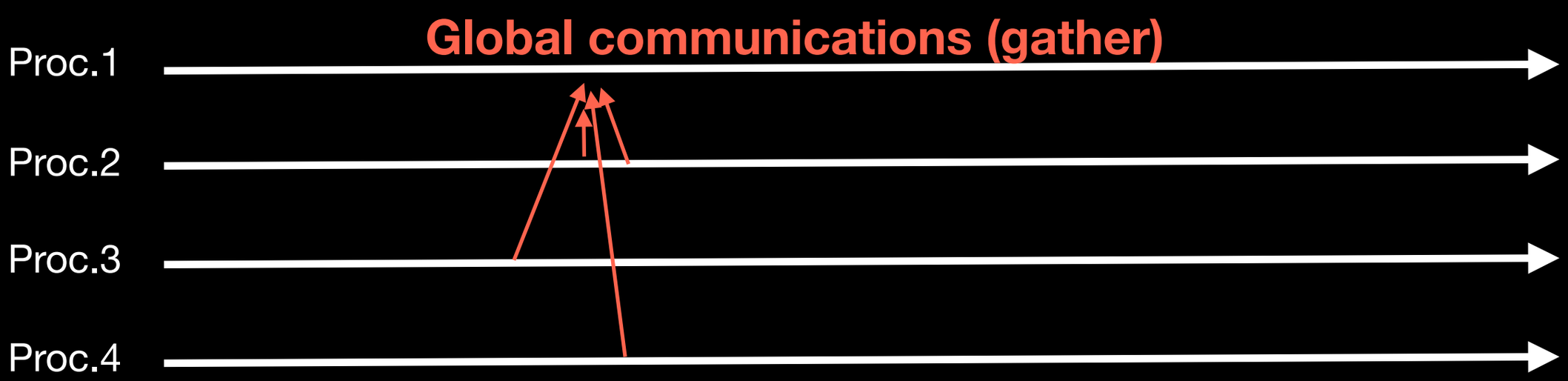


Distributed parallelism

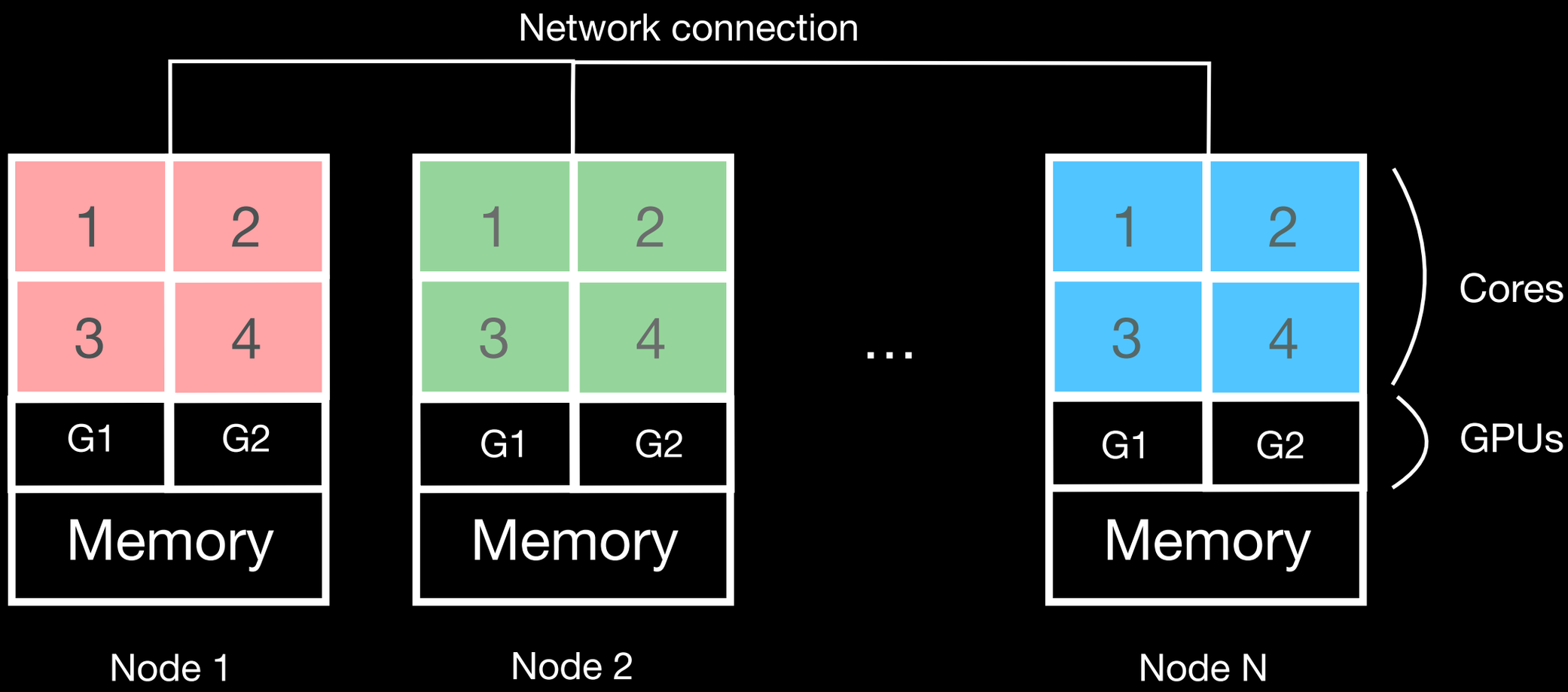


Distributed parallelism





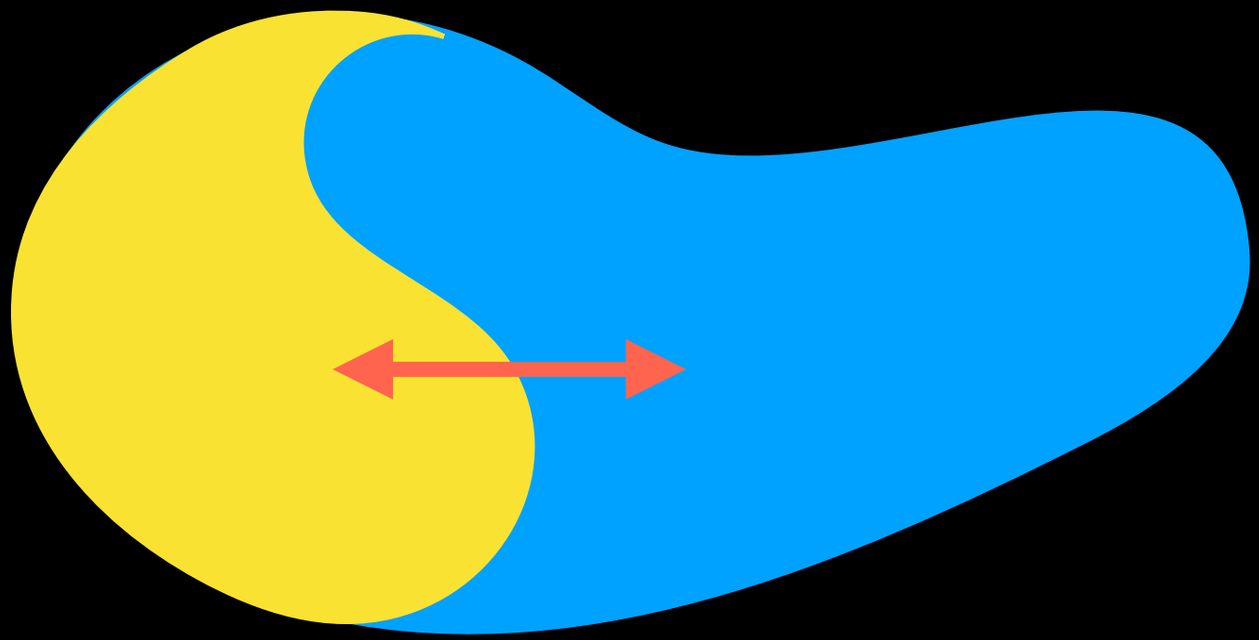
Distributed parallelism



- Explicit code to implement communications across different processes
- communications have a cost (per com + per byte)
- global communications do not scale

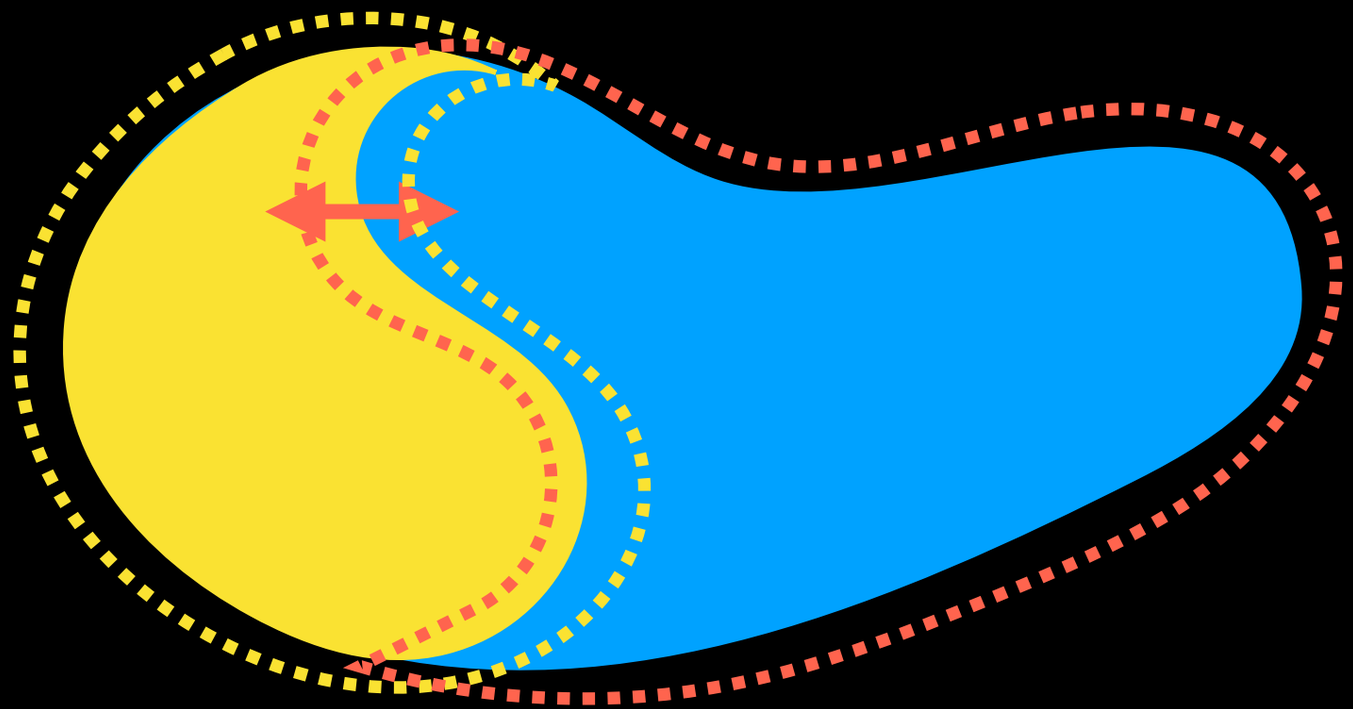
How to?
- Message Passing Interface (MPI) library is de facto a standard in HPC
- Standard protocol, different implementations (MPICH, OpenMPI, IntelMPI,…)

What is parallel?



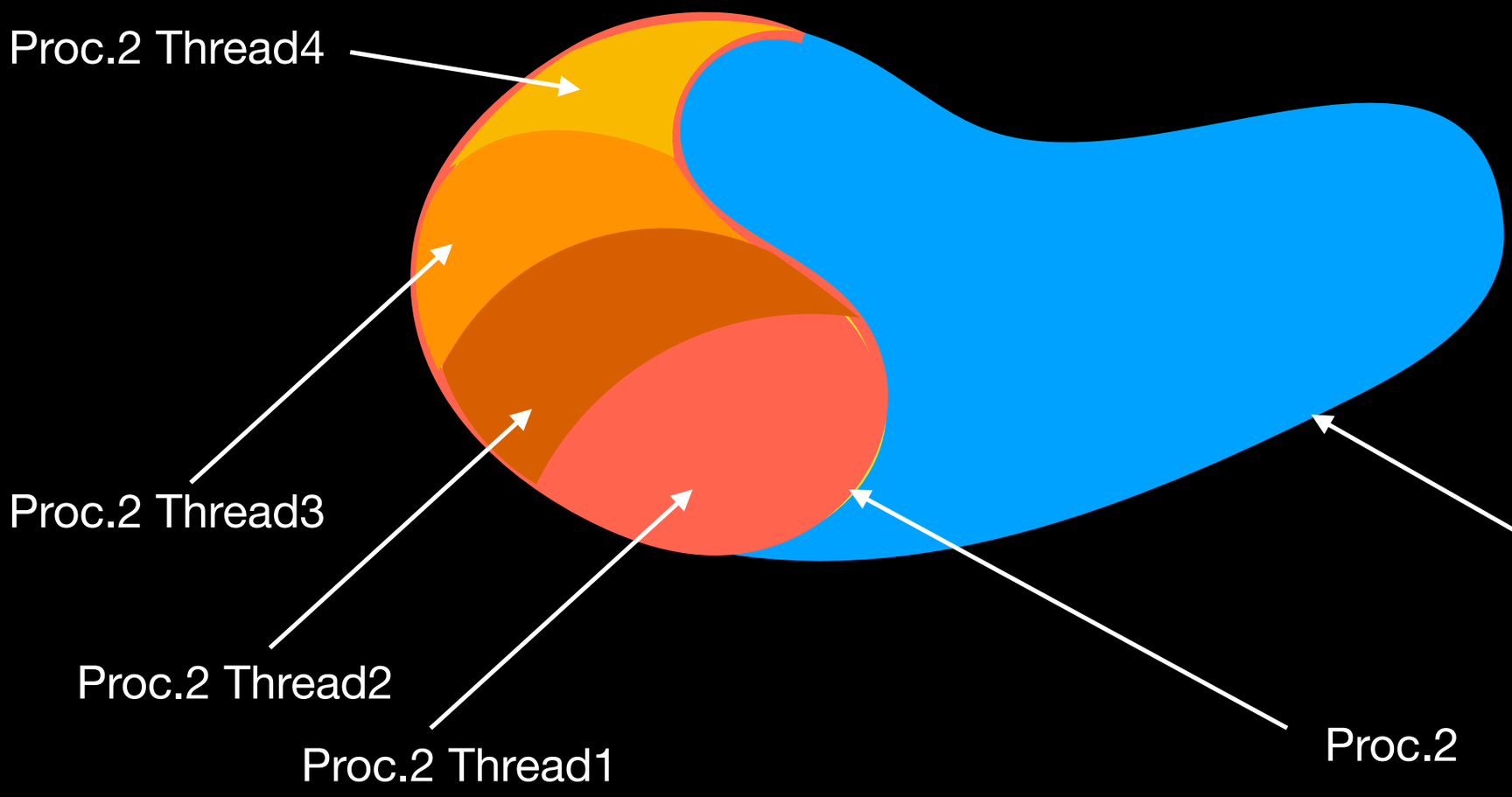
- Cut the workload in N processes
- Each process computes the equations on its part of the work load
- They « synchronize » before going to next step

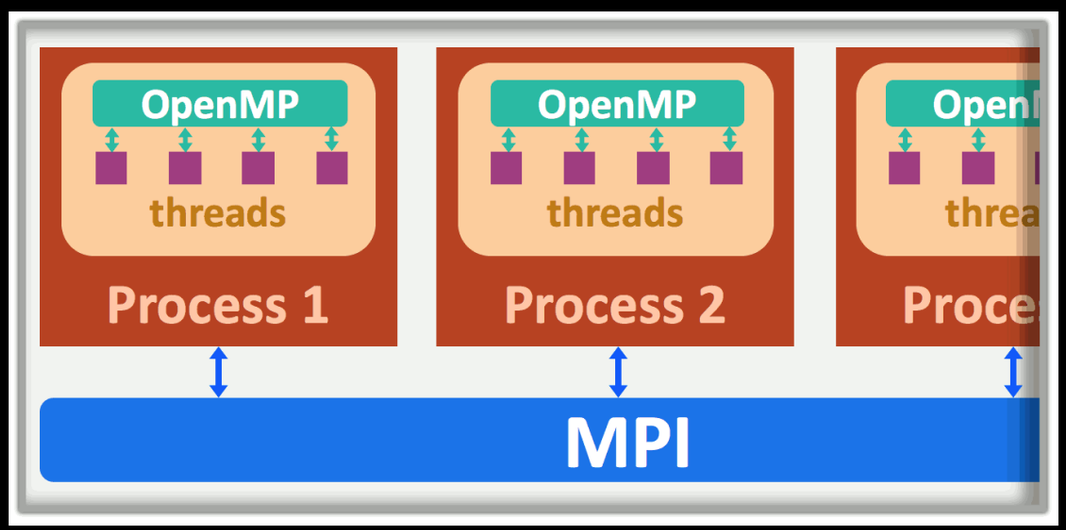
Distributed parallelism



- Cut the workload in N processes
- Each process computes the equations on its part of the work load
- They « synchronize » before going to next step, « ghost » regions (copy from overlapped neighbor domain)

Hybrid thread+process (distributed/shared) parallelism

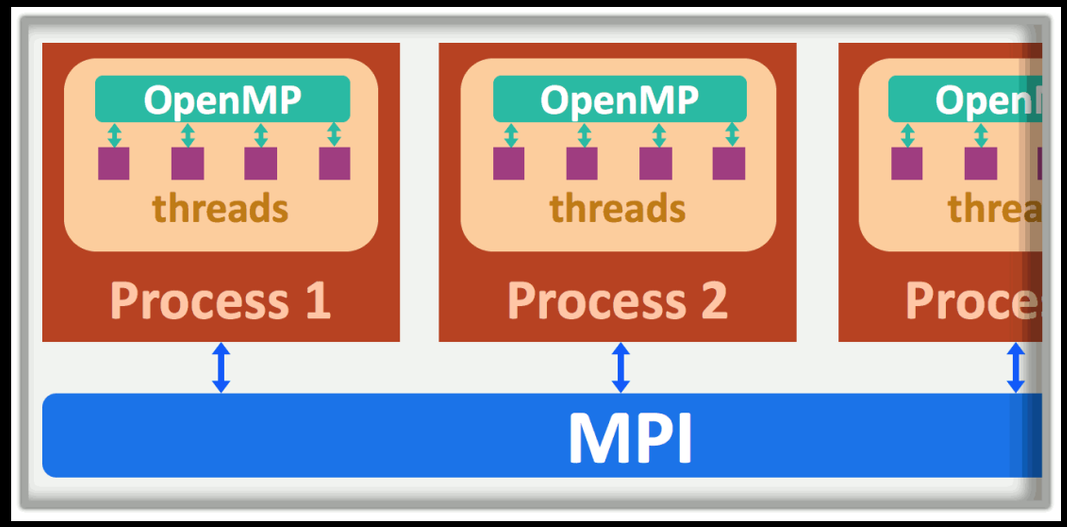




(From https://smileipic.github.io/Smilei/)

HYBRID THREAD+PROCESS (DISTRIBUTED/SHARED) PARALLELISM

- Take advantage that cores on the same node share memory : no communication overhead
- Is more complex because you now deal with two parallelization paradigms possibly requiring adjustments to your data structures (what's optimal for MPI may not be for threads and vice versa)
- Hybrid thread+MPI parallelization generally comes after *full MPI implementations*



(From https://smileipic.github.io/Smilei/)

Scalable parallelism

Scalability is the property of your program to continue taking advantage of more parallelism

The depressing Amdahl's law

T total runtime in serial, T(p) runtime on p cores ,s: sequential fraction, f: parallel fraction

$$S_a = \frac{T}{T(p)} = \frac{T}{sT + f\frac{T}{p}} = \frac{1}{1 - f + \frac{f}{p}} \qquad \lim_{p\to\infty} S_a = \frac{1}{s}$$

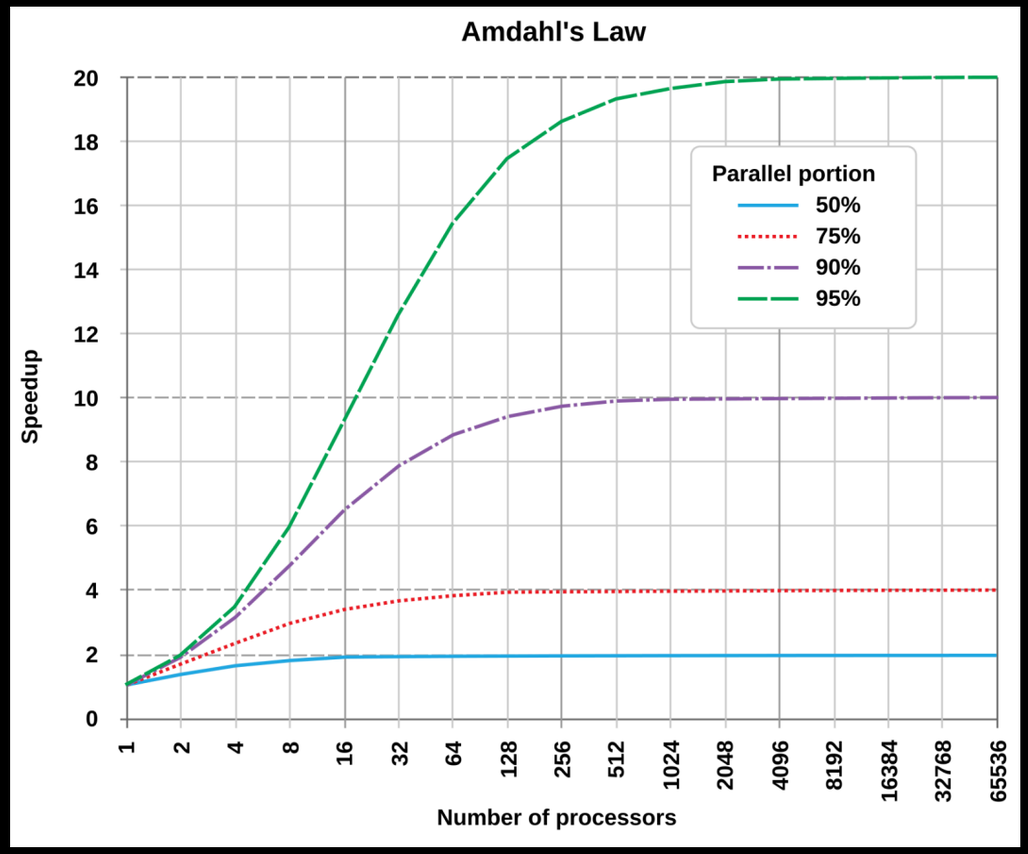
Amdahl's law tells you how the runtime changes when the total work load is constant and the number of cores increases. The serial fraction kills scalability.

The depressing Amdahl's law

Assume $f = 95\%$

$$S_a = \frac{1}{1 - f + \frac{f}{p}}$$

$$S_a(100) \approx 17$$



GUSTAFSON'S LAW

Assume you increase the work load *proportionally* to the number or cores. Parallel time is T=1

How much slower the serial version will be if a fraction s of the code is serial?

$$S_g = \frac{T}{T(p)} = \frac{s + pf}{1} = 1 - f + pf = s - (s-1)p$$

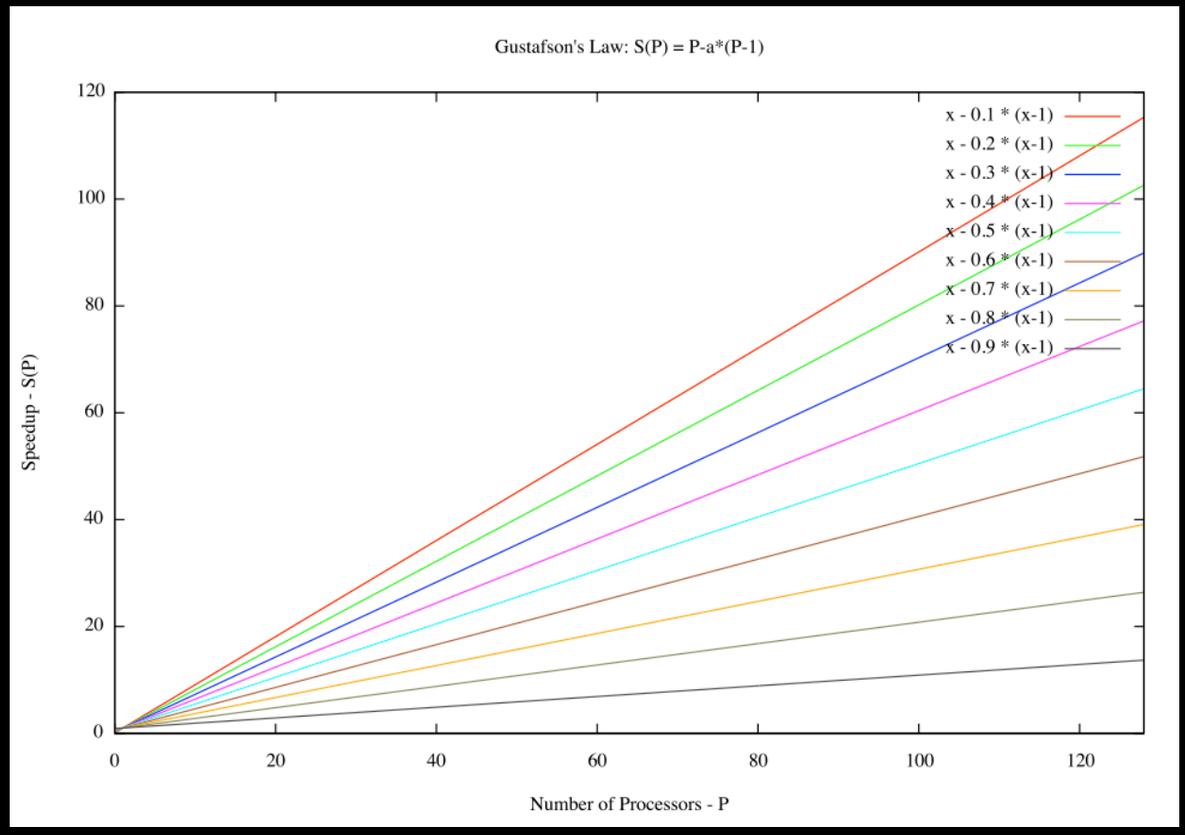
Gustafson's law tells you how you can benefit from parallelization for solving a better problem.

GUSTAFSON'S LAW

Assume $f = 95\%$

$$S_g = s - (s-1)p$$

$$S_g(100) \approx 95$$

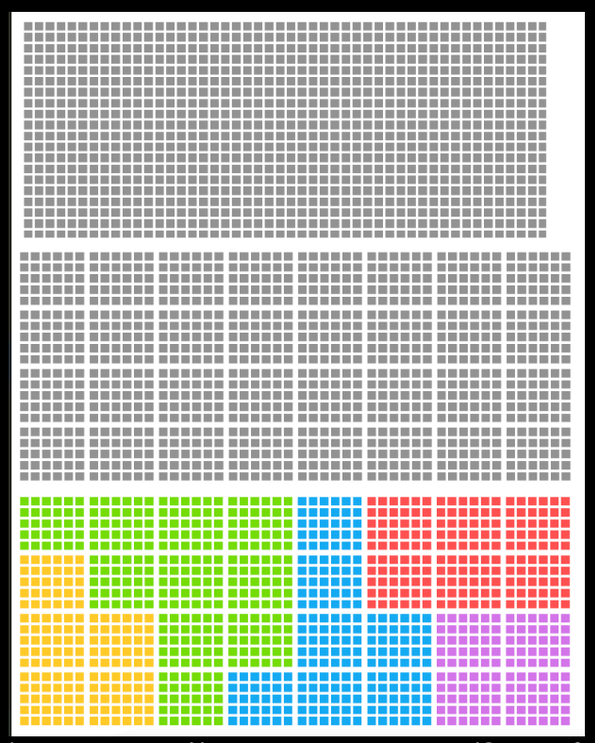


Neither Amdahl's or Gustafson's law are right/wrong. They are guidelines as to how your parallelization behaves depending on what you want to achieve.

Testing Amdahl's law : *strong* scaling test

Testing Gustafson's: *weak* scaling test

Load Balancing



(From https://smileipic.github.io/Smilei/)

Example of some strategy (from SMILEI PIC code)

Cut the spatial domain in many regular rectangular « patches »

How do you distribute those patches across MPI processes?

Some patches have many particles… some few

Equal number of patches per process will not **balance the work load**

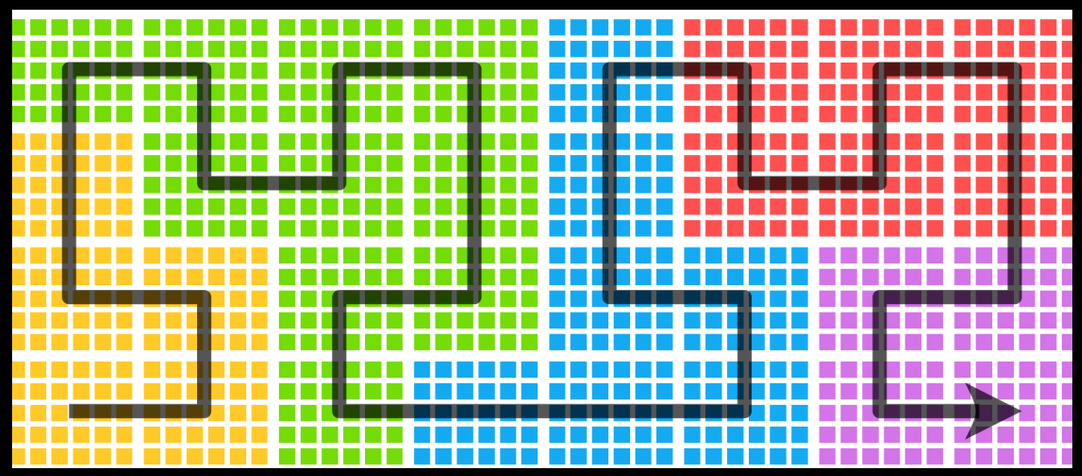
SCALABILITY

You need a way to give about the same work load to each process

SMILEI : Hilbert Space Filling Curve, fractal 1D line that passes through each of the patches

« periodically » assess unbalance growth (from particle motion)

Cut the line in N segments of approximately equal work load

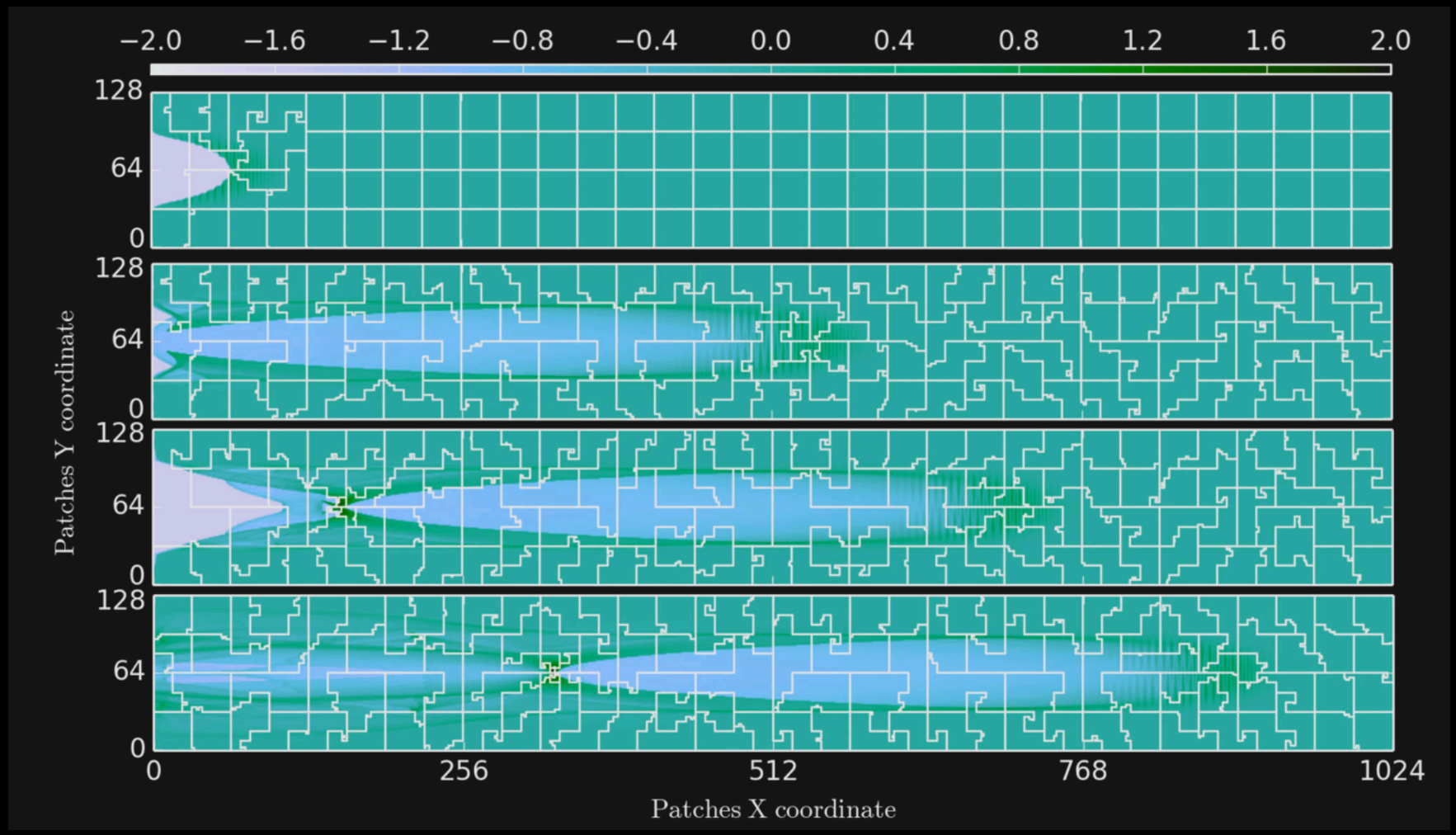


(From https://smileipic.github.io/Smilei/)

Dynamic Load Balancing the Particle computations

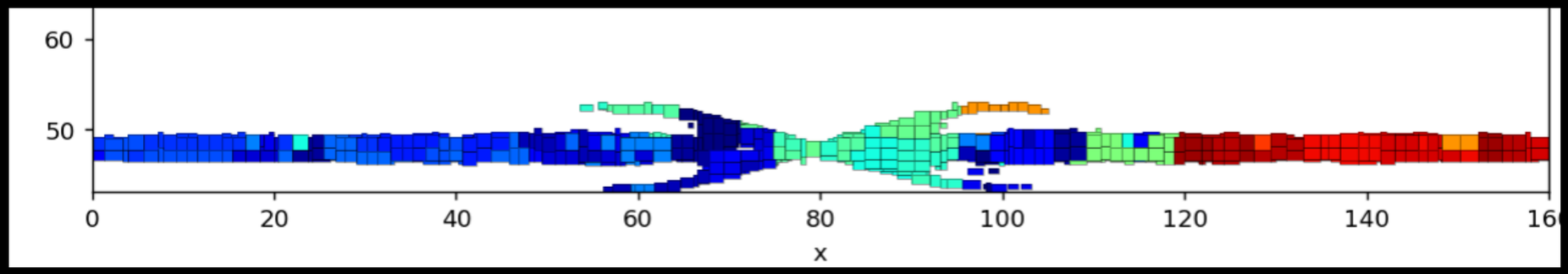
Example in SMILEI simulations

time

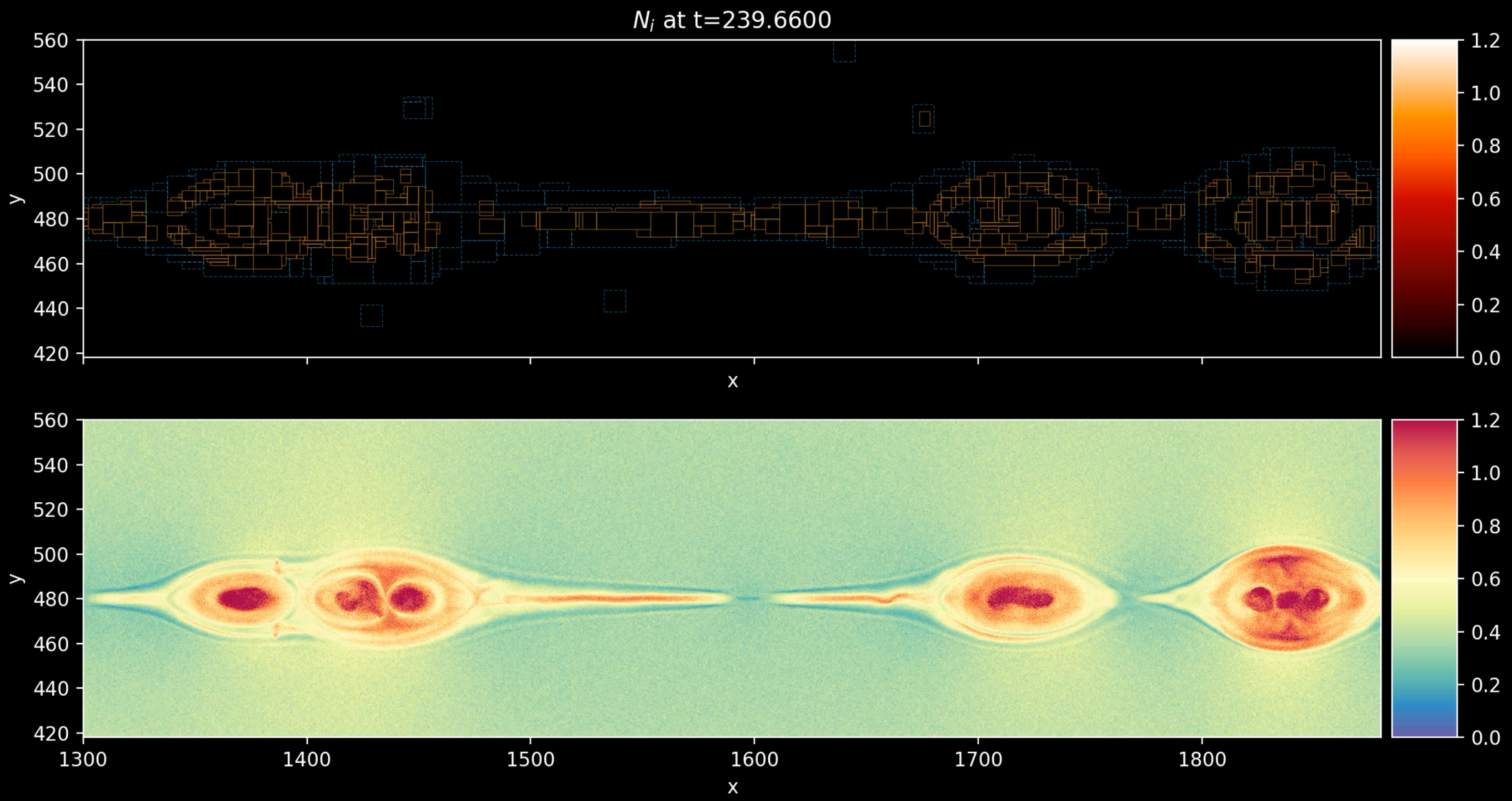


(From https://smileipic.github.io/Smilei/)

Dynamic Load Balancing Adaptive Mesh Refinement



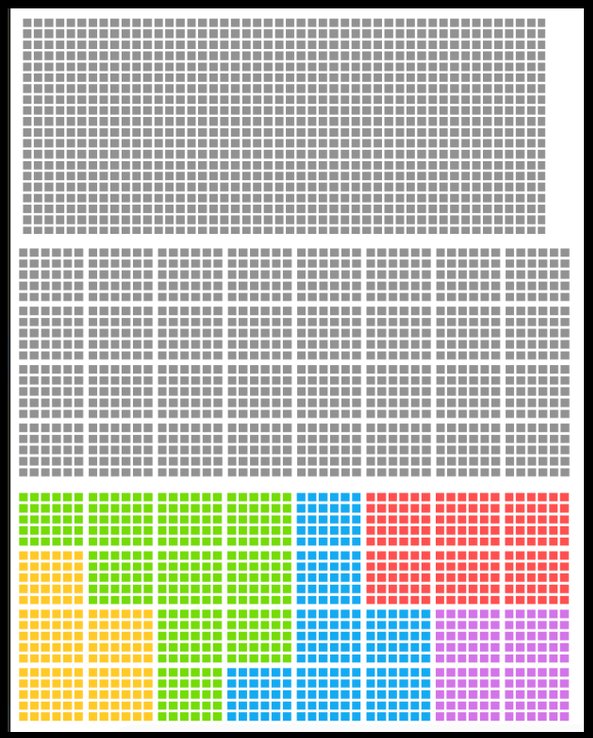
(From https://github.com/pharehub/phare)

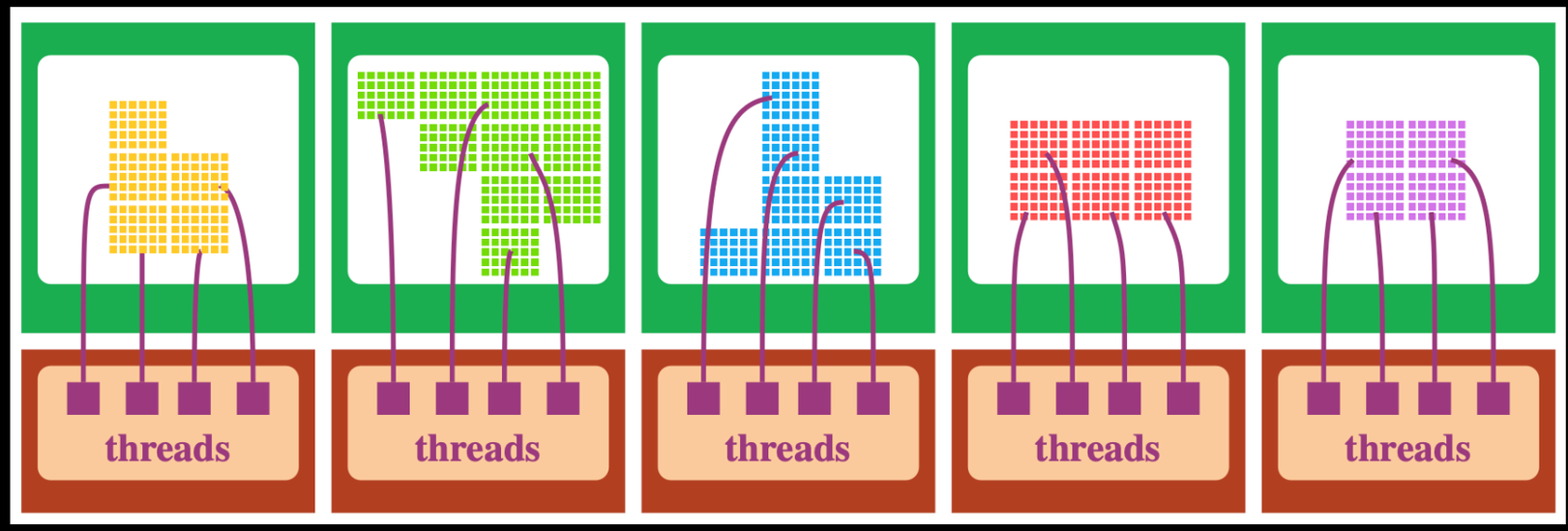


MPI + CPU THREADS

If you have many patches per process and you treat them sequentially you waste time

Handle local patches with N threads





(From https://smileipic.github.io/Smilei/)

Load balance also applies to how work load is dispatched across threads…

You don’t want 1 thread to do everything while the other wait…