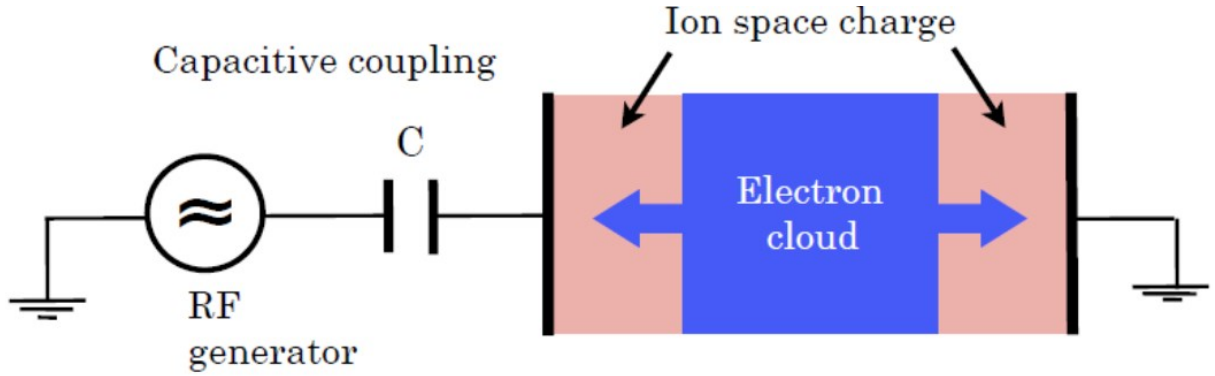
Implementation strategies for Multi-GPU PIC/MCC plasma simulation on pre-exascale systems

Zoltan Juhasz[1], Zoltan Donko[2] and Peter Hartmann[2]

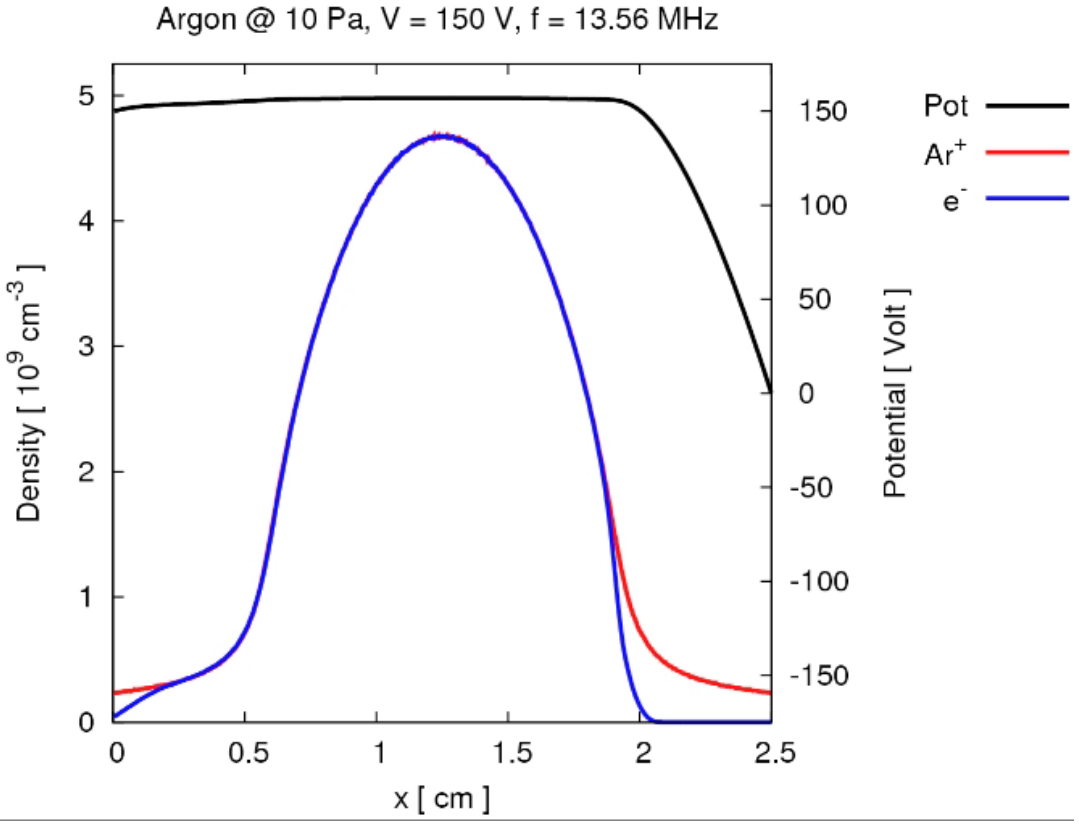
[1]Dept. of Electrical Engineering and Information Systems,
University of Pannonia, Veszprem, Hungary

[2]Dept. of Complex Fluids, Institute for Solid State Physics and Optics,
Wigner Research Centre for Physics, Budapest, Hungary

Plasma simulation

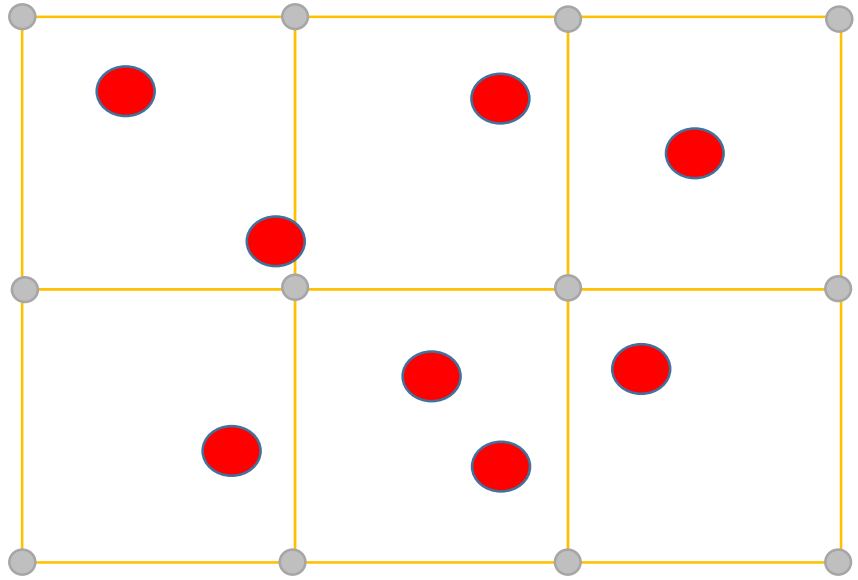


- Understanding capacitively coupled radiofrequency discharges in plasma
- Spatiotemporal changes in electric field
- Non-equilibrium transport of particles
- Numerical simulation helps to understand the behaviour of particles
- Uses kinetic theory for describing particle movement
- 1D and 2D geometries

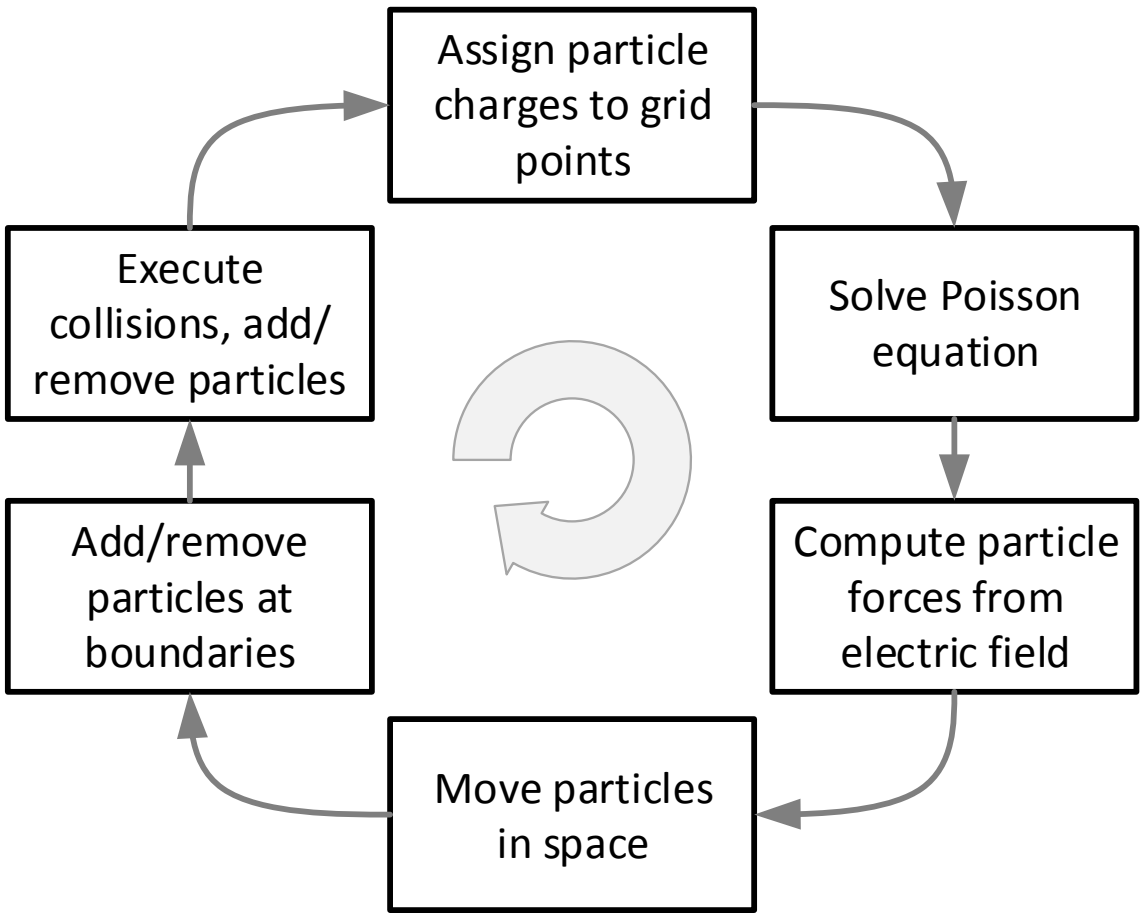


Approach: Particle-in-Cell (PIC) simulation

- N-body problem
  - no direct particle interaction, particles interact with the field
  - place particle charges to grid
  - solve grid for field – Poisson equation
  - move particles based on field forces
- Particle count: from 100k to 10m particles
- Complication: collision!
- CPU execution is long: ranging from days to months
  - parallel solutions: OpenMP and/or MPI code
  - irregular memory accesses make code difficult to parallelise efficiently



Baseline: single-GPU implementation (1D geometry)



Loop for simulation cycles (1000-3000)

Loop for input samples (800)

1. move electrons -- e_move kernel
2. check boundaries -- e_boundary kernel
3. electron collision -- e_collisions kernel
4. electron density calculation -- e_density kernel
5. move ions -- ion_move kernel
6. check boundaries -- ion_boundary kernel
7. ion collision -- ion_collisions kernel
8. ion density calculation -- ion_density kernel
9. Poisson solver -- CPU seq. solver (Thomas algorithm)

Problems:

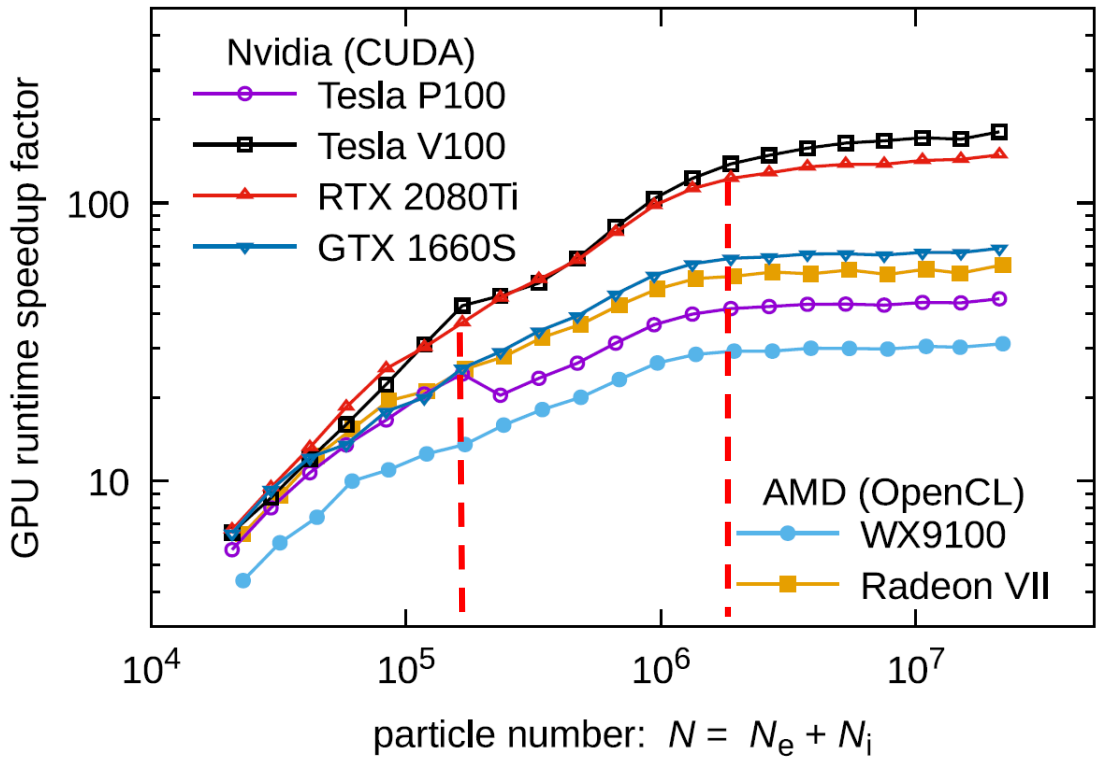
- too many small kernels with low op. intensity,
- memory bound kernels,
- kernel launch overhead,
- CPU Poisson solver, host-device data transfer

GPU PIC/MCC behaviour

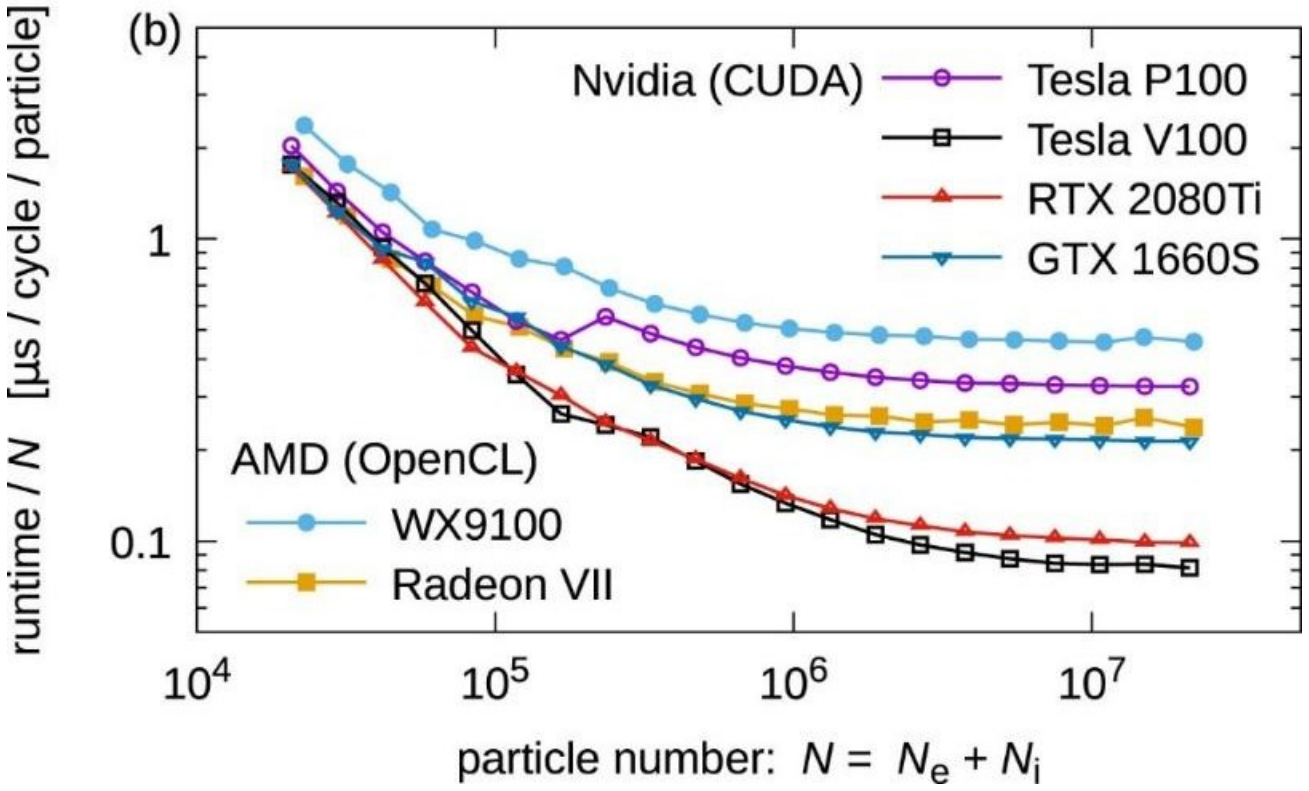
Multi-GPU advantage:

- introduce **further speedup** for a system of given size (strong scaling)
- **increase particle count** without increasing execution time (weak scaling)

Speedup



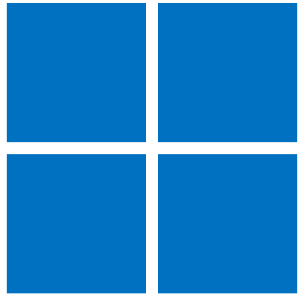
per-particle execution time



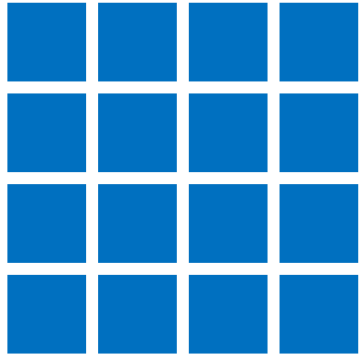
Typical GPU systems (desktops, small clusters)

1 node
1 A100 GPU
6,912 cores

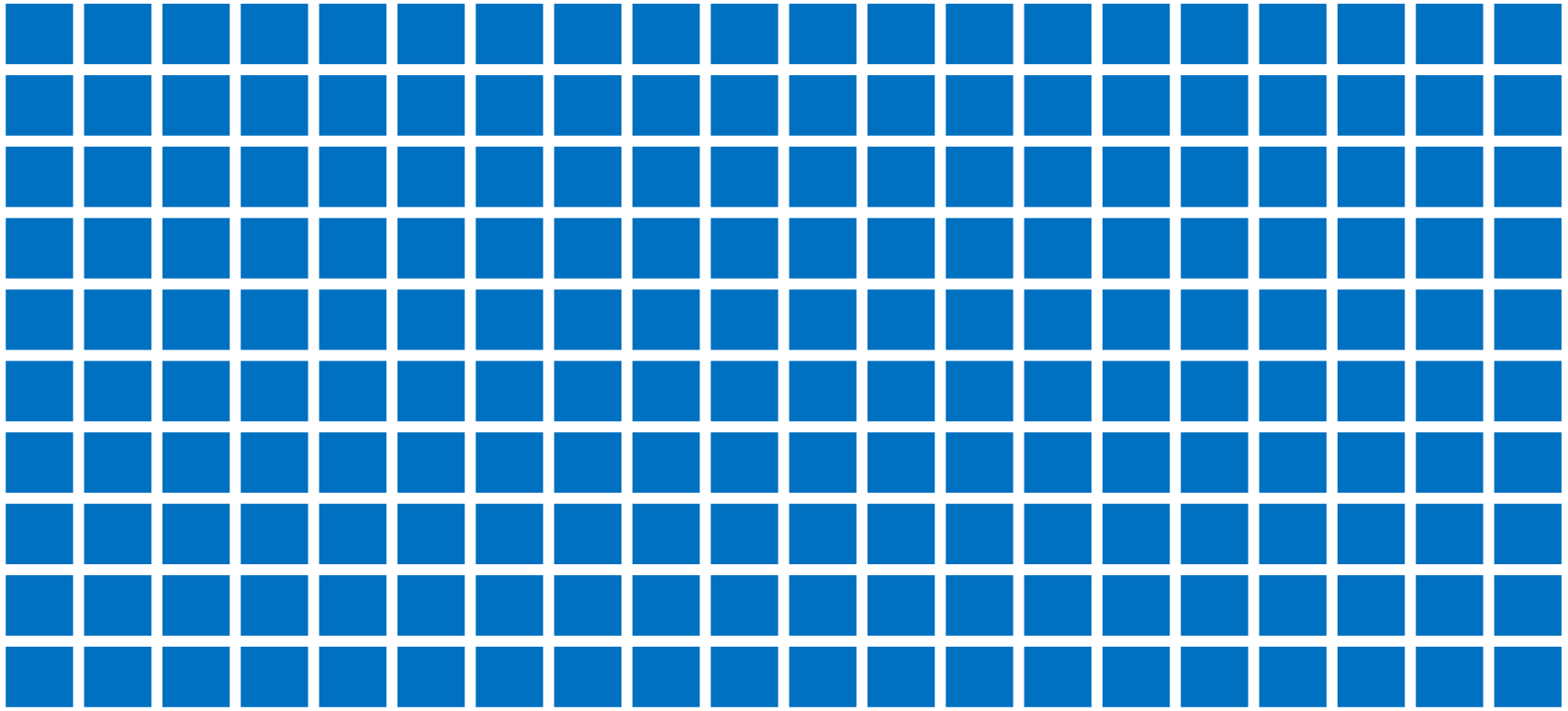
1 node
4 A100 GPUs
27,648 cores



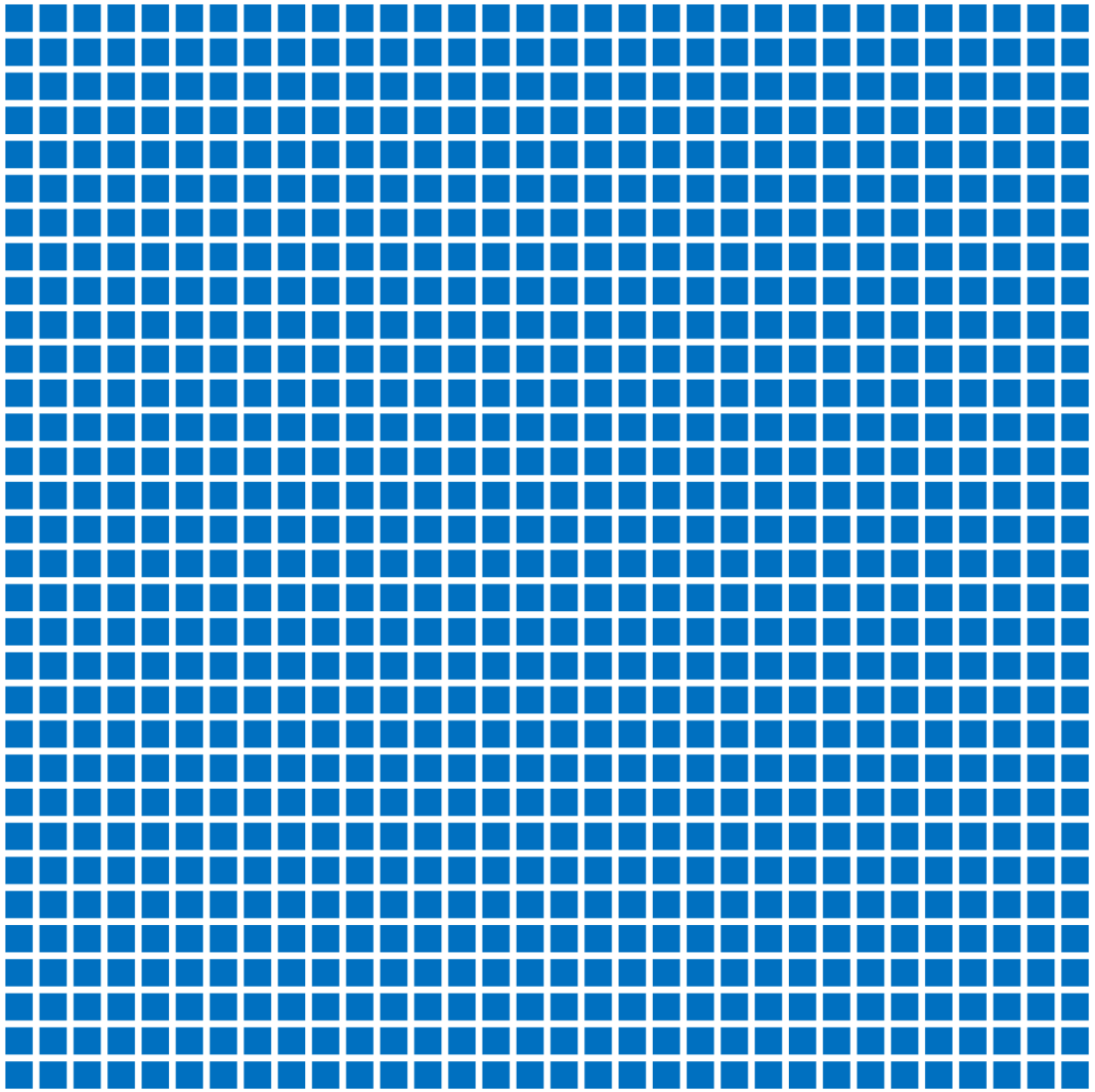
4 nodes
16 A100 GPUs
110,592 cores



Komondor (Hungary, coming soon):
200 GPUs, 4.5 Pflop/s



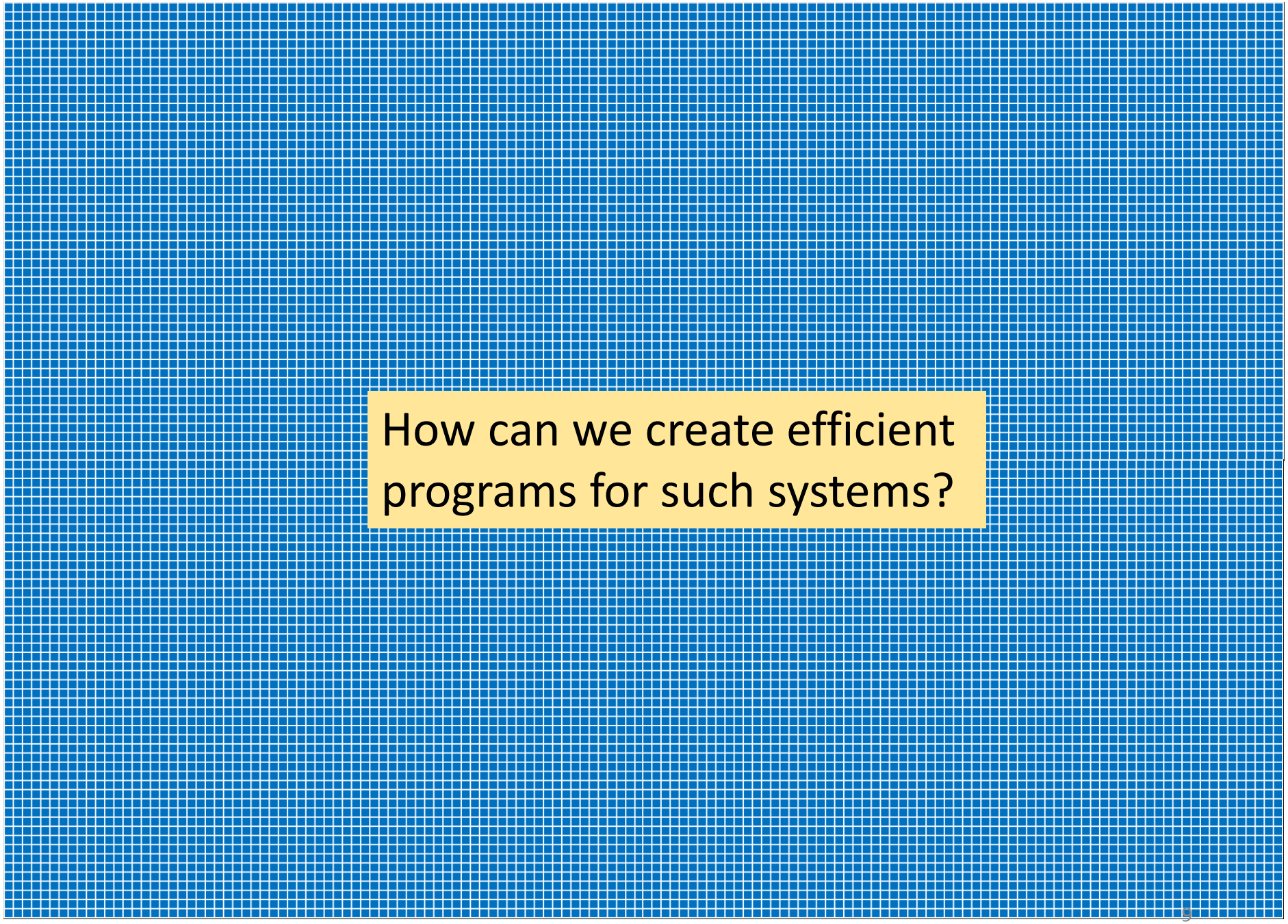
Marconi 100 (Italy):
256 out of 3920 GPUs
32 Pflop/s



Leonardo (Italy):

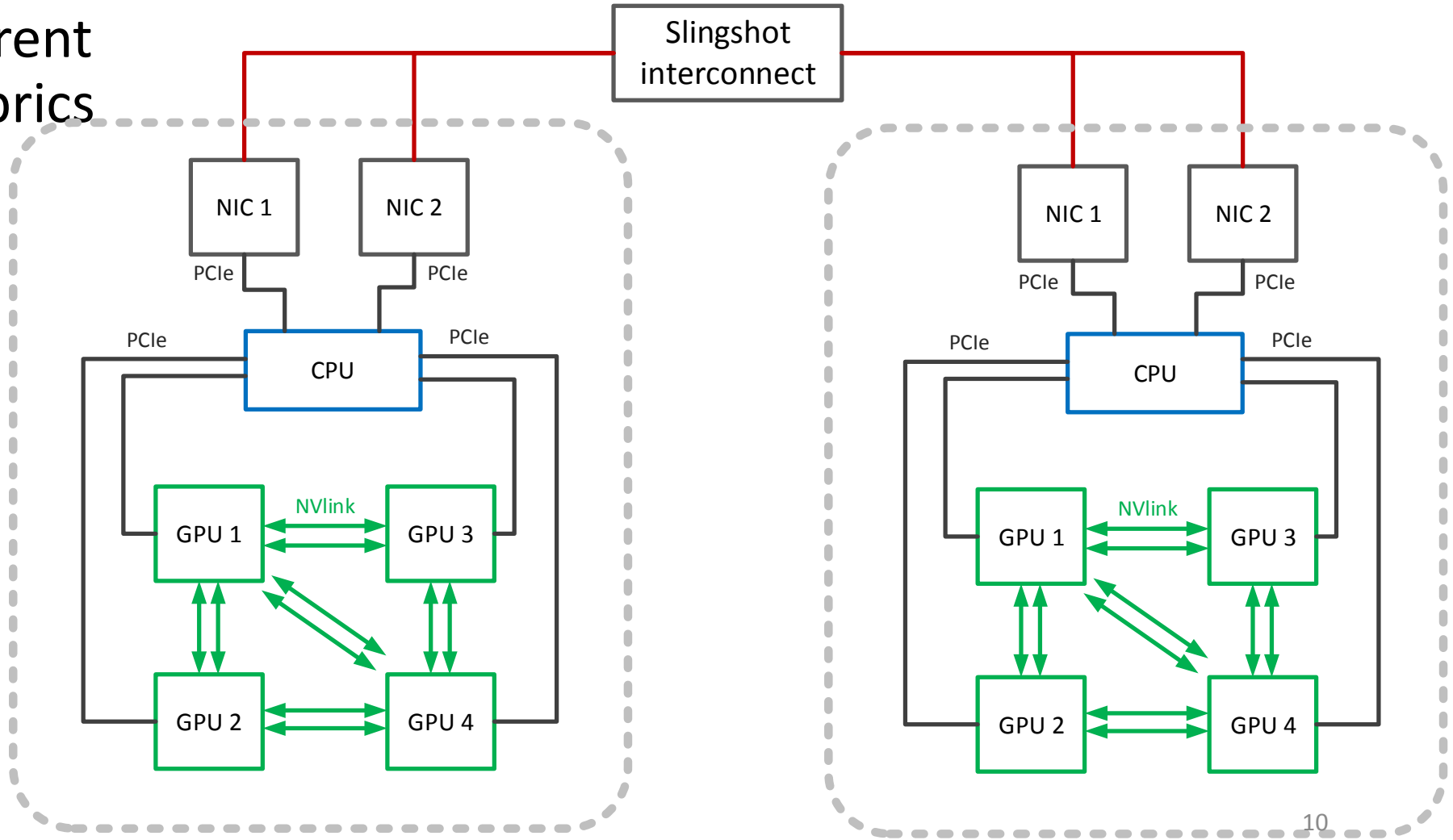
14000 GPUs
200+ Pflop/s

by end of 2022



Pre-exascale multi-GPU architecture

- At least three different communication fabrics
  - PCI-e (CPU-GPU)
  - Nvlink (GPU-GPU)
  - Slingshot or Infiniband (node-node)
- HPE Cray EX
- MARCONI 100
- LEONARDO



Multi-GPU plasma simulation program strategies

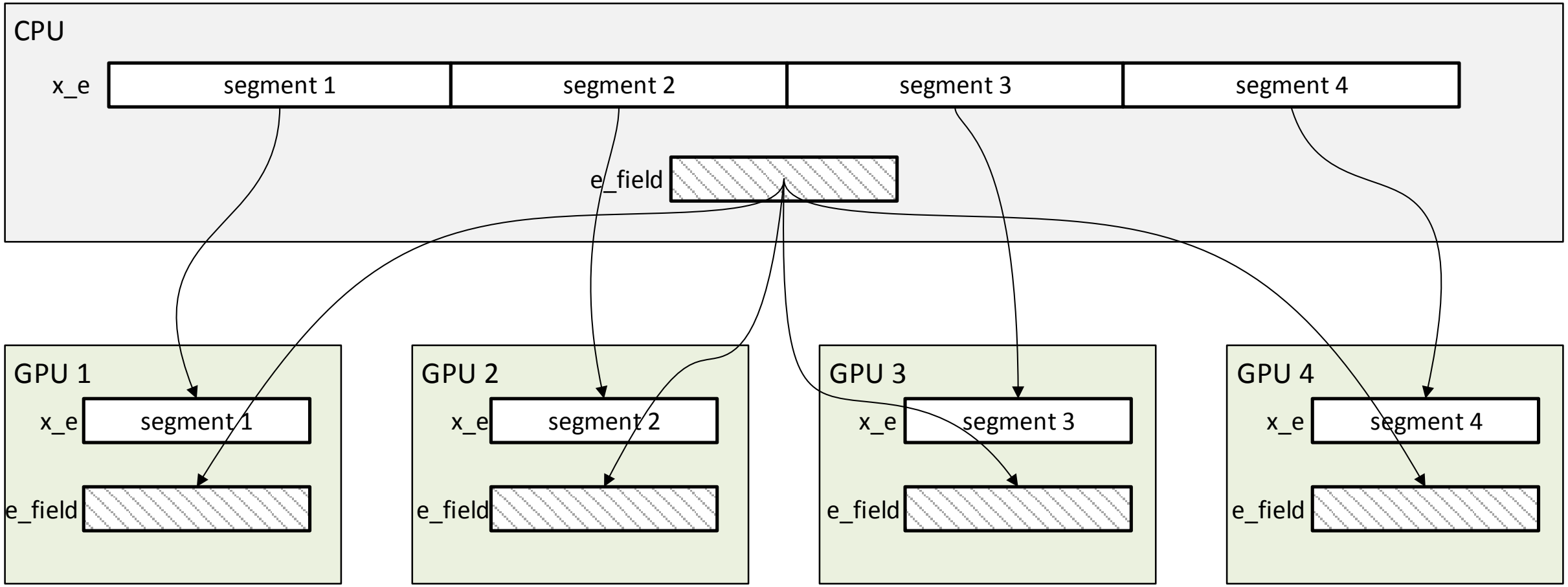
- Simulation strategies
  - **domain decomposition**: each cell of the grid on one GPU
  - **particle decomposition**: particles are distributed over the GPUs
- Implementation strategies
- single node, single thread
- single node, multithread (OpenMP)
- multi-node case
  - MPI
  - OpenMP/MPI hybrid
  - OpenMP/CUDA-aware MPI hybrid
  - NCCL
  - NVSHMEM

Single node, single thread multi-GPU program

```
// distributing the workload across multiple devices
for (int i = 0; i < ngpus; i++) {
    cudaSetDevice(i);
    cudaMemcpyAsync(d_A[i], h_A[i], iBytes,
                    cudaMemcpyHostToDevice, stream[i]);
    cudaMemcpyAsync(d_B[i], h_B[i], iBytes,
                    cudaMemcpyHostToDevice, stream[i]);
    kernel<<<grid, block, 0, stream[i]>>> (d_A[i], d_B[i], d_C[i], iSize);
    cudaMemcpyAsync(h_C[i], d_C[i], iBytes,
                    cudaMemcpyDeviceToHost, stream[i]);
}
cudaDeviceSynchronize();
```

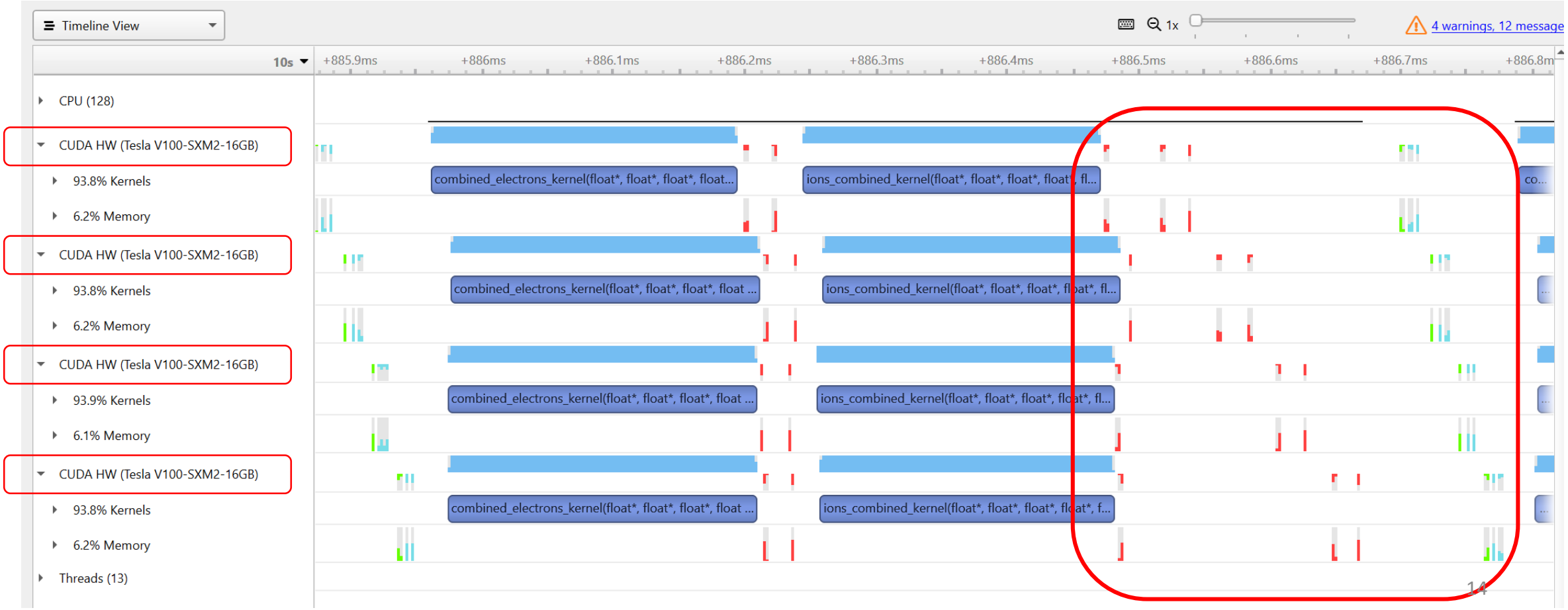
Data distribution

- each GPU moves particles locally and computes new charge densities
- Poisson solver on CPU or on each GPU (requires collective communication)

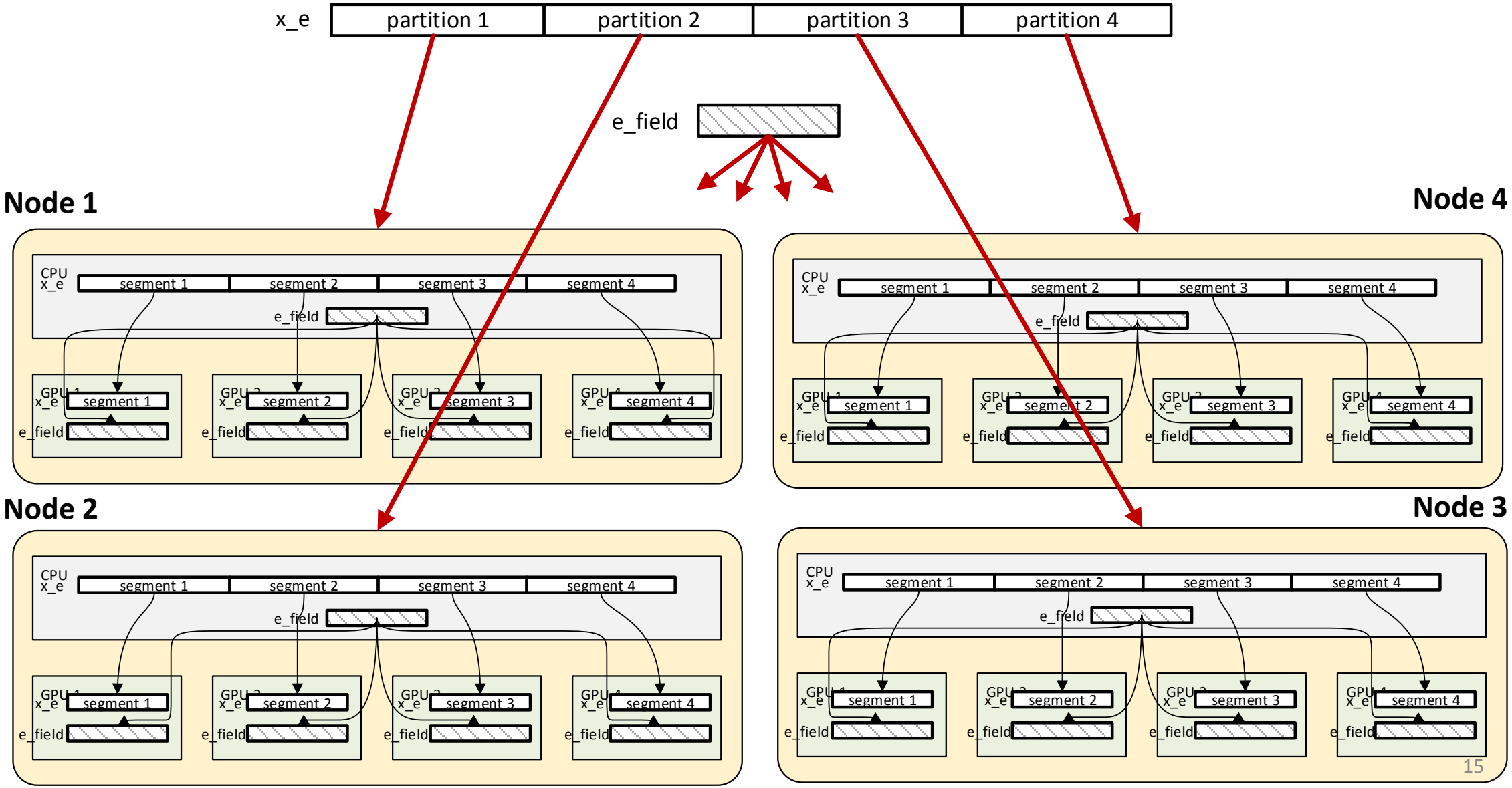


Multi-GPU 1D execution, single node, OpenMP

- execution timeline of a single iteration step



Multi-node case, MPI strategy



Problems

- Requires collective communication operations involving CPU and GPU memory
  - traditional MPI is not suitable – requires extra copy operations from GPU to CPU
  - CUDA-aware MPI is OK, can use GPU pointers
- Communication initiation is on CPU
  - difficult to overlap comms. with GPU kernel execution
  - communication can eventually become a performance bottleneck
  - scalability limit

Alternative No 1 – NCCL

- NVIDIA Collective Communication Library
- Provides uniform API for all-gather, all-reduce, broadcast, reduce, reduce-scatter, point-to-point send and receive comm. routines
- Optimized to achieve high bandwidth and low latency over PCIe, NVLink and other high-speed interconnects
- Automatic topology detection for high bandwidth paths on AMD, ARM, PCI Gen4 and InfiniBand
- Supports multi-threaded and multi-process applications
- InfiniBand, RoCE and IP Socket internode communication
- NCCL 2.4 introduced tree-based all-reduce instead of rings

Mismatch of communication libraries and the CUDA model

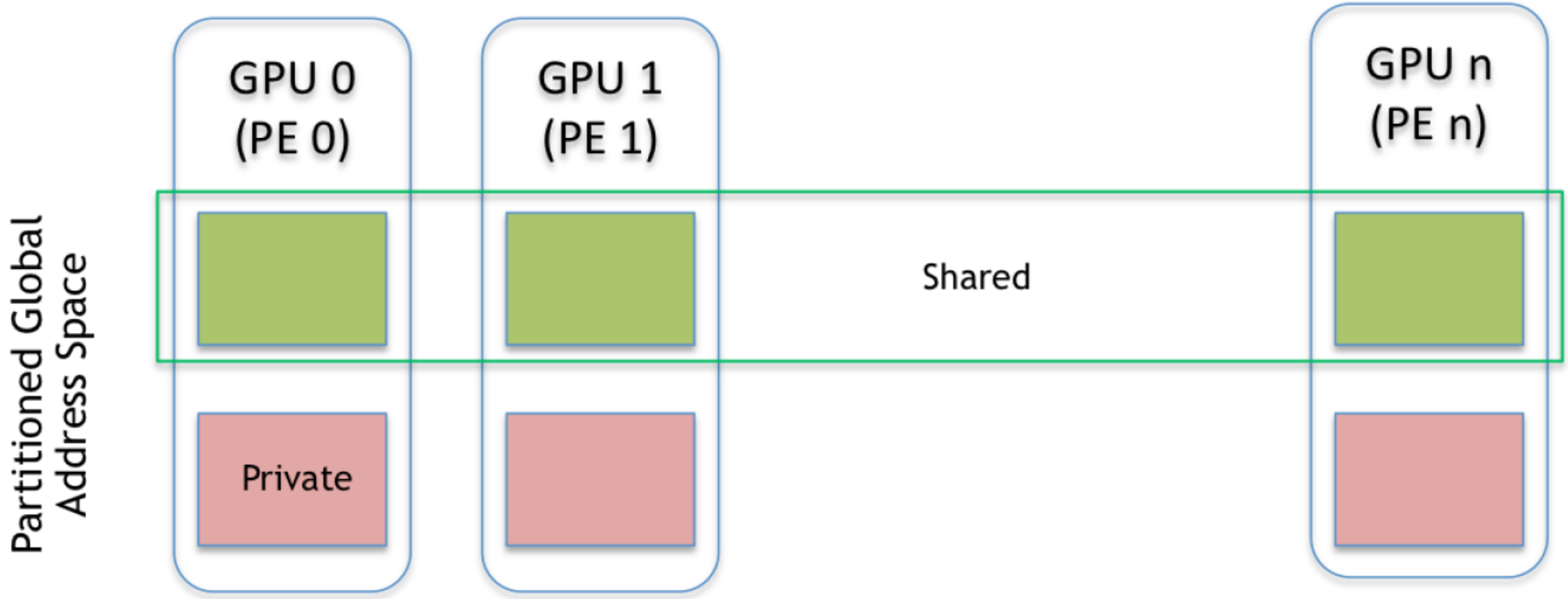
- MPI and NCCL provide CPU-initiated communication routines
- Assumes separate computation and communication steps
- Fine for synchronous execution where all processes execute in parallel
- CUDA model relies on large number of threads to hide memory and instruction latency: threads >> cores
- Internal scheduling of kernels (thread blocks) makes it difficult to hide communication
- Need a distributed model more aligned with the CUDA programming model

Alternative No 2 – NVSHMEM

- NVIDIA OpenSHMEM Library implementation – development started for and with the SUMMIT supercomputer system
- Virtual shared memory system
- Collective and point-to-point communication routines called from CPU or from GPU kernel
- GPU-initiated communication: much simpler overlap of communication with computation
- Provides:
  - Remote memory access (RMA: PUT/GET)
  - Atomic memory operations (AMO)
  - Signal operations
  - Direct load and store operations
  - Collective functions (broadcast, reductions, and others)
  - Wait and test functions (local symmetric memory only)

Partitioned Global Address Space

- Private vs global memory space
- private: normal device memory
- global: {symmetric address, PE index} pair



NVSHMEM-based plasma code version

- kernels move/collide particles locally on each GPU
- update density vectors
- solve for local fields on each GPU
- perform **all-reduce** to update global e_field OR update global e_field with NVSHMEM **atomicAdd** intructions in overlapped fashion

2D geometry

- 1D case
  - particle count up to 1-10 millions: 1k cells, 1-10k particles per cell
- 2D case
  - 500 x 500 or 1k x 1k grid
  - 100 to 1k particles per cell
  - 25 M – 1 B particles
- e_field should be in shared memory on each GPU
  - 1M instead of 1K elements
  - Ampere persistent shared memory could help, otherwise must be moved into global memory

Summary

- Still in progress, single node and rudimentary MPI versions are complete, NVSHMEM version coming soon
- There are many alternatives for structuring and implementing pre-exascale multi-GPU programs
- Selecting the best strategy and creating efficient code might not be easy
- Following implementation patterns developed for much smaller CPU-based systems might not be a good long-term choice
- Do not be afraid to re-design, re-structure your program
- Programs with hundreds of millions or billions of threads can be executed efficiently on state-of-the-art systems