



# U.S. Particle Accelerator School

Education in Beam Physics and Accelerator Technology

*Self-Consistent Simulations of Beam and Plasma Systems*

Steven M. Lund, Jean-Luc Vay, Rémi Lehe and Daniel Winklehner

Colorado State U., Ft. Collins, CO, 13-17 June, 2016

## A4. Intro to Parallel Computing

Remi Lehe

Lawrence Berkeley National Laboratory

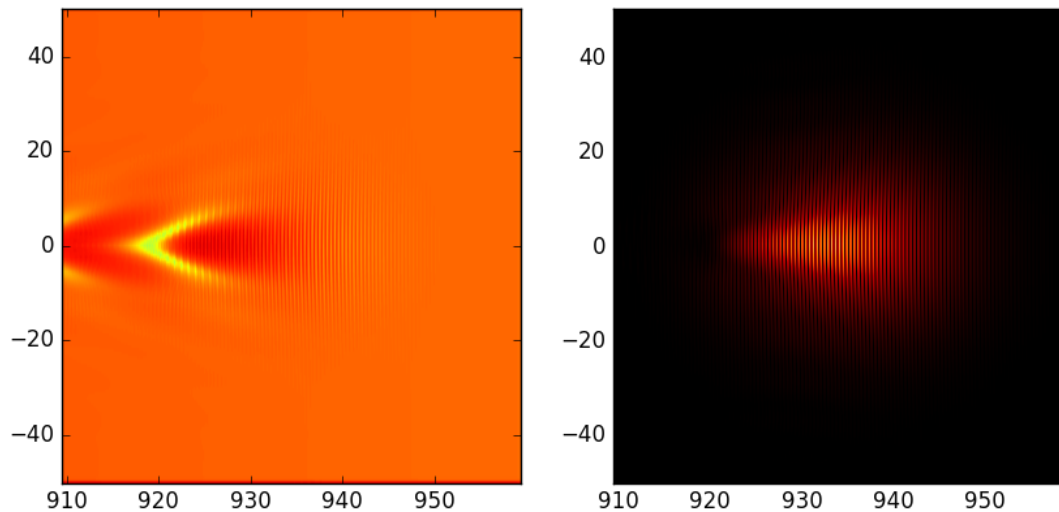
# Outline

- Modern parallel architectures
- Parallelization between nodes: MPI
- Parallelization within one node: OpenMP



# Why use parallel architecture?

- **Recent example: Laser-wakefield simulation to interpret experiments at LBNL.**



3D grid with 2500 x 200 x 200 grid points  
0.2 billion macroparticles  
140,000 timesteps (Courant limit!)

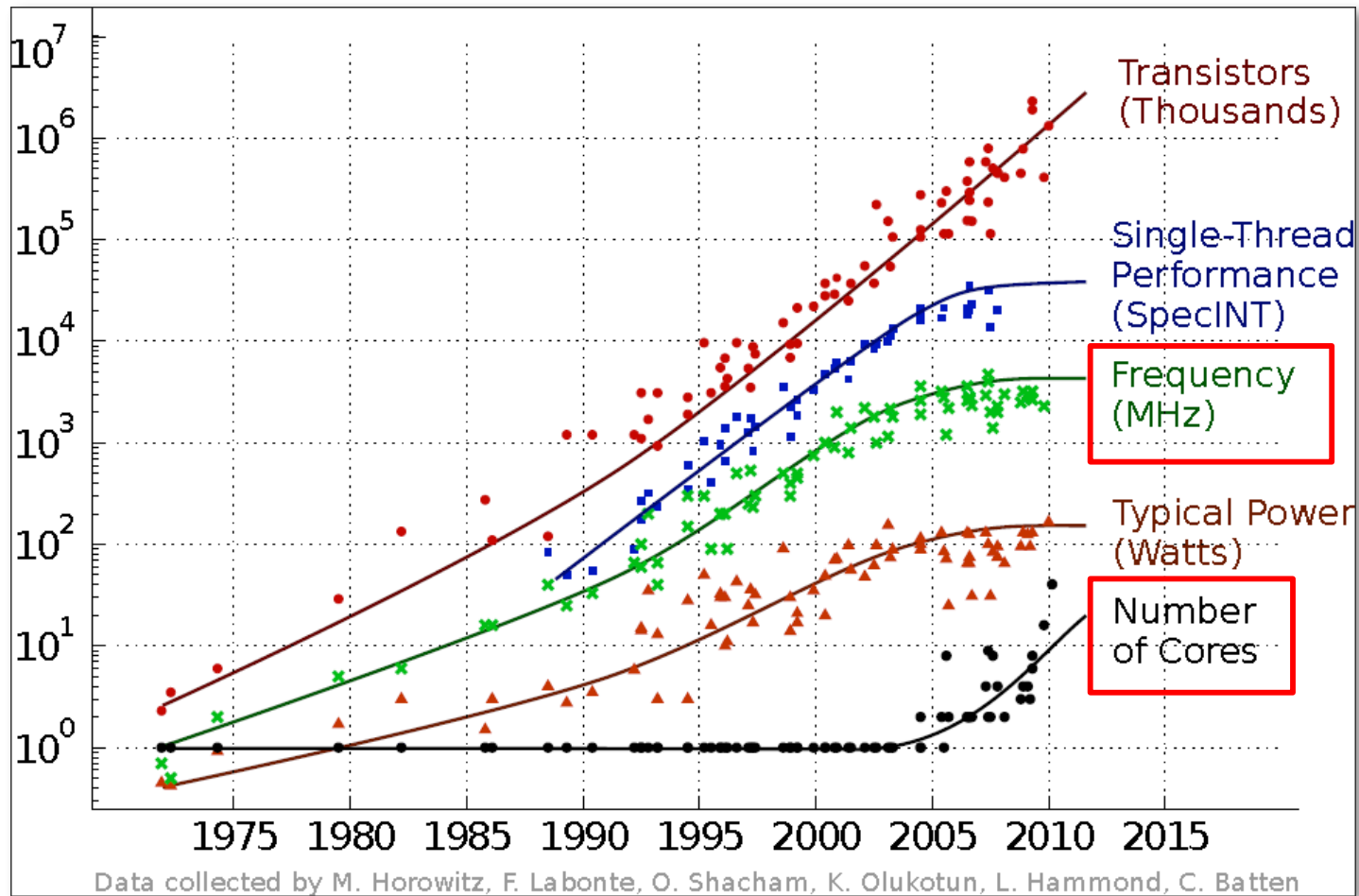
~60,000 hours on 1 core = 7 years!

=> Need either faster cores or more cores in parallel



# Why use parallel architectures?

## History of CPU performance



Nowadays, individual cores do not get faster.  
We need to use many cores in parallel.



# Parallel clusters

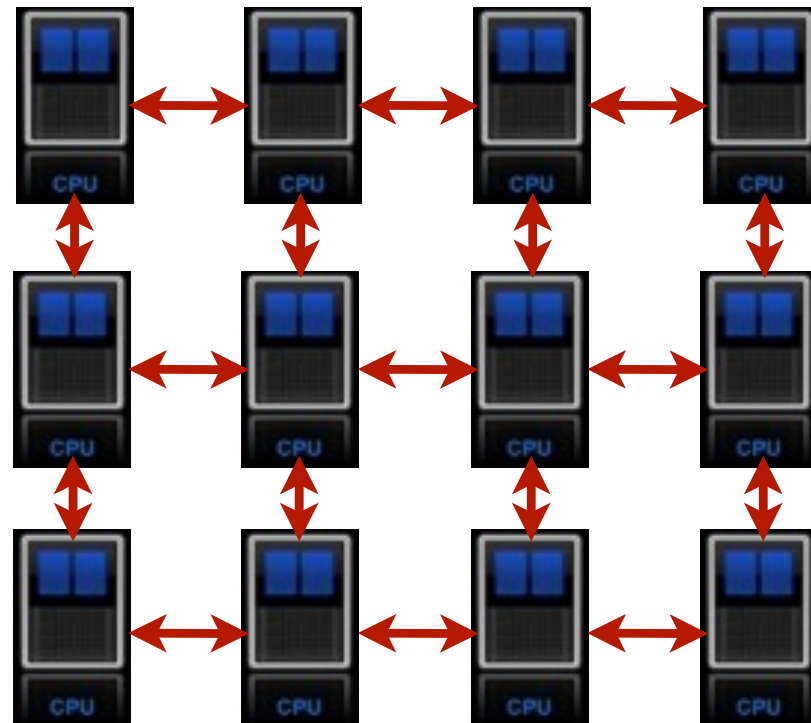


Contain 100,000+ cores

↔ “Fast” network communication (~10 Gb/s)



Individual node



# Leadership and Production Computing Facilities



## Titan:

- Peak performance of 27.1 PF
- 18,688 Hybrid Compute Nodes
- 8.9 MW peak power



## Mira:

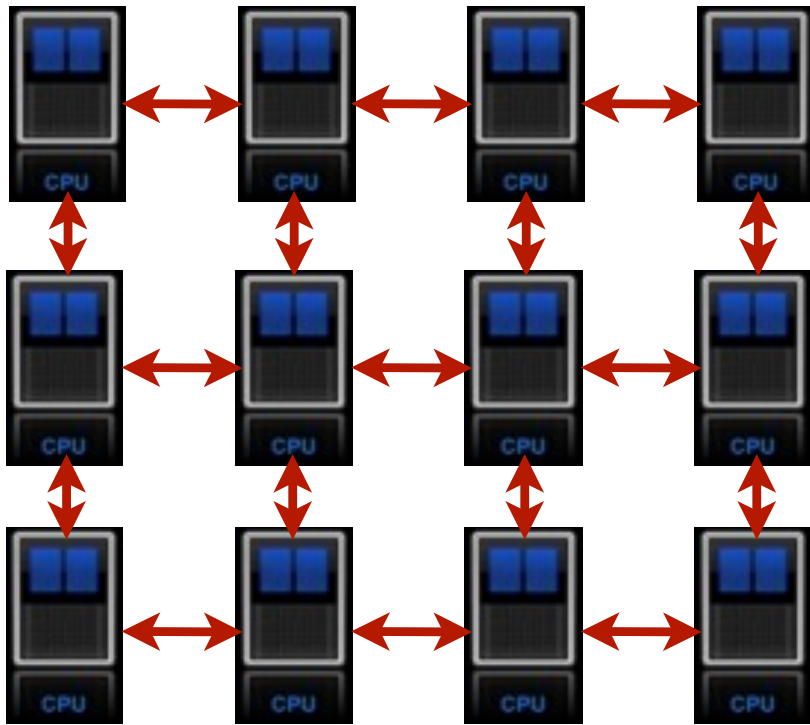
- Peak performance of 10 PF
- 49,152 Compute Nodes
- 4.8 MW peak power

## Edison XC30:

- Peak performance 2.4 PF
- 124,608 processing cores
- 2.1 MW peak power



# Parallel clusters



Individual nodes have several cores (i.e. computing units):

- “Traditional” CPUs: ~10 cores
- Xeon Phi: 68 cores
- GPUs: ~1000 (slow) cores

Cores within one node share memory

- How to use this architecture to make a PIC code faster?
- How to use the two levels of parallelism?  
(within one node and between nodes)



# Outline

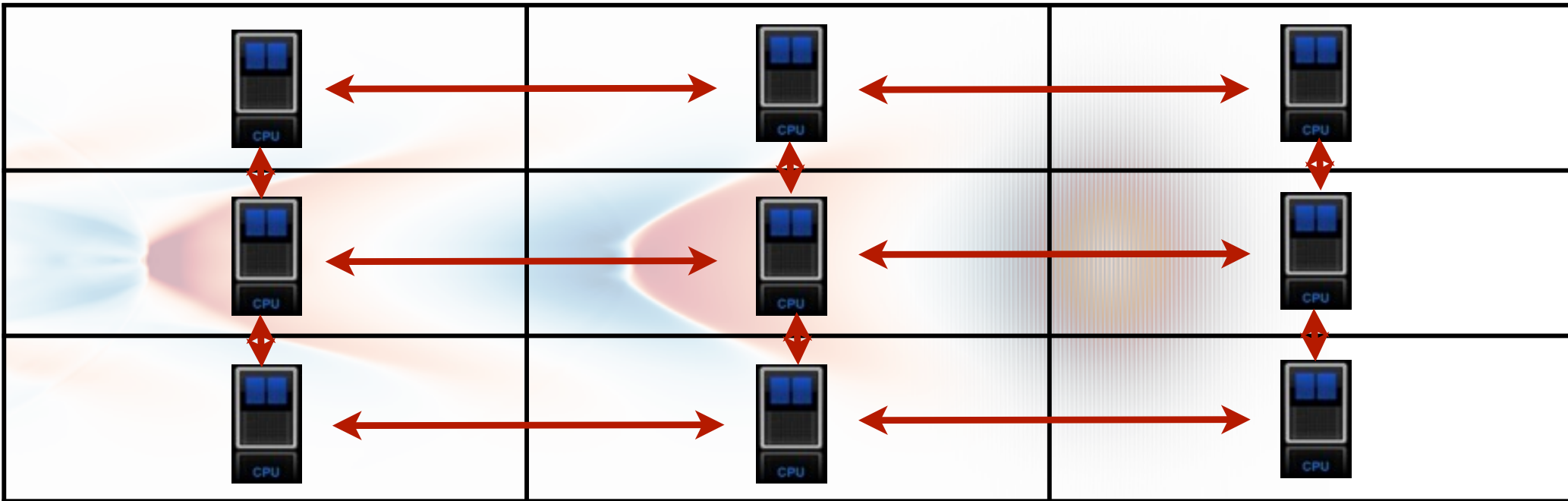
- Modern parallel architectures
- Parallelization between nodes: MPI
- Parallelization within one node: OpenMP





# Domain-decomposition

Each node deals with a fixed chunk of the simulation box  
(which includes fields on the grid + macroparticles)



↔ “Fast” network communication

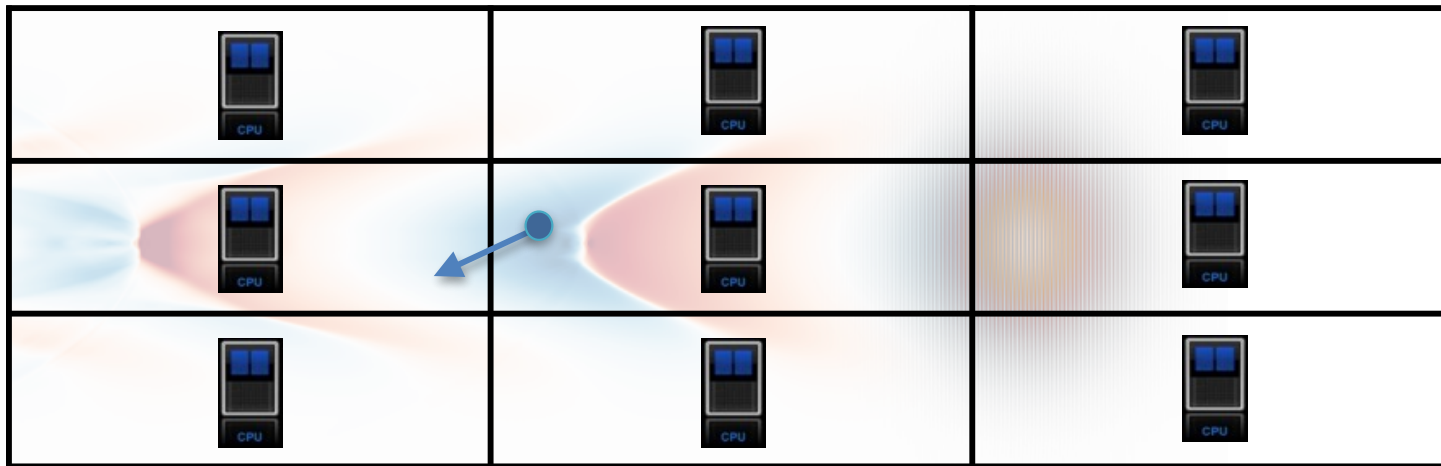
**The nodes are not independent:**  
They need to exchange information with other nodes via the **network**.

Domain-decomposition minimizes this:  
communications only with a few neighbors



# Domain-decomposition: particle exchange

Particle pusher: macroparticles may cross domain boundaries



After the particle pusher, the particle data needs to be communicated from one node to the other.

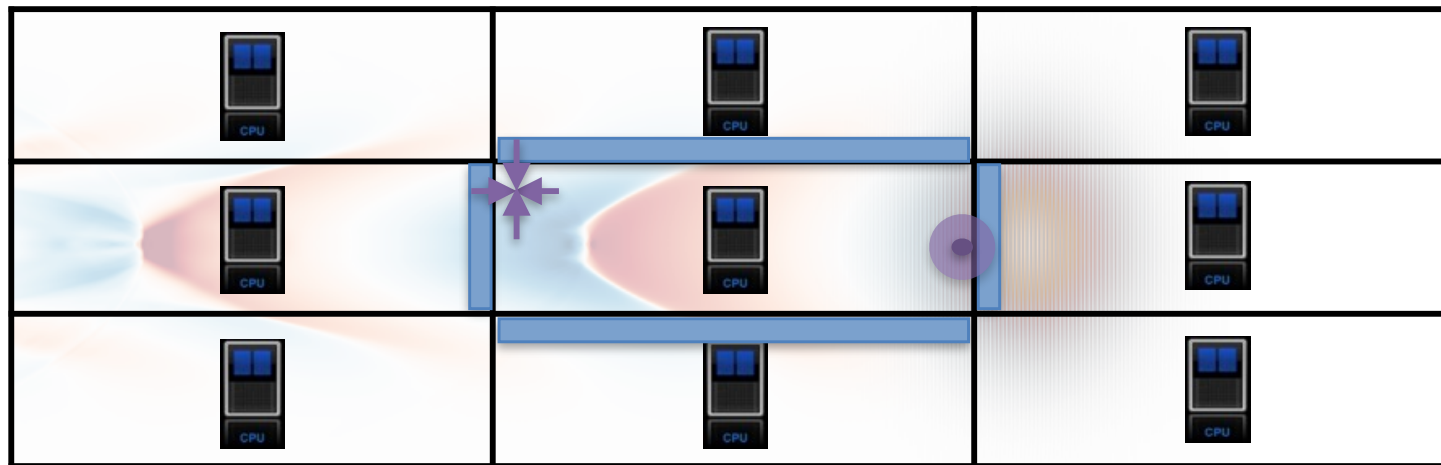


# Domain-decomposition: field exchange

**Field update:** each node needs values from neighboring nodes to calculate the spatial derivatives at the boundary

$$\partial_t B_y|_{i+\frac{1}{2},j,k+\frac{1}{2}}^n = -\partial_z E_x|_{i+\frac{1}{2},j,k+\frac{1}{2}}^n + \partial_x E_z|_{i+\frac{1}{2},j,k+\frac{1}{2}}^n$$

**Field gathering/Current deposition:** with wide shape factor, particles gather field values from other nodes and deposit some current/charge to other nodes

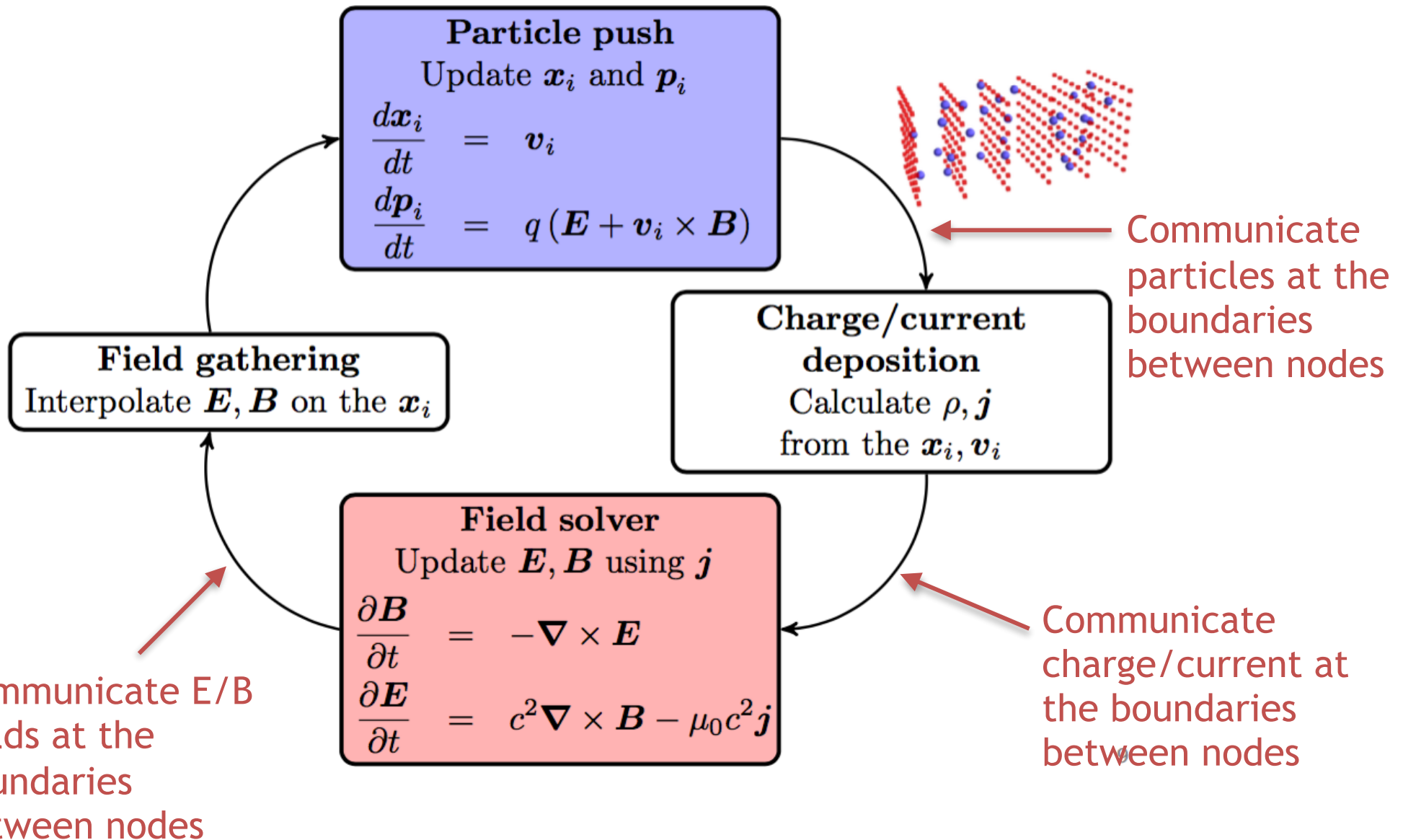


**After the current deposition:** communicate the current that has been deposited into other nodes

**After the field update:** communicate the new values of the fields at the border of the box



# Sum up: the PIC loop

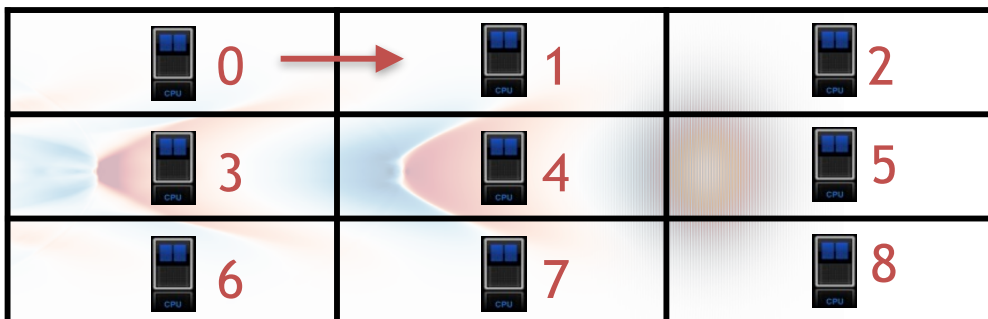


# Exchanging information: MPI

## MPI: Message Passing Interface

Library that allows to send/receive information between processes:

- Each process has an id (or “rank”)
- Each process executes the **same source code**
- Call sending/receiving commands based on id



## Example in Python:

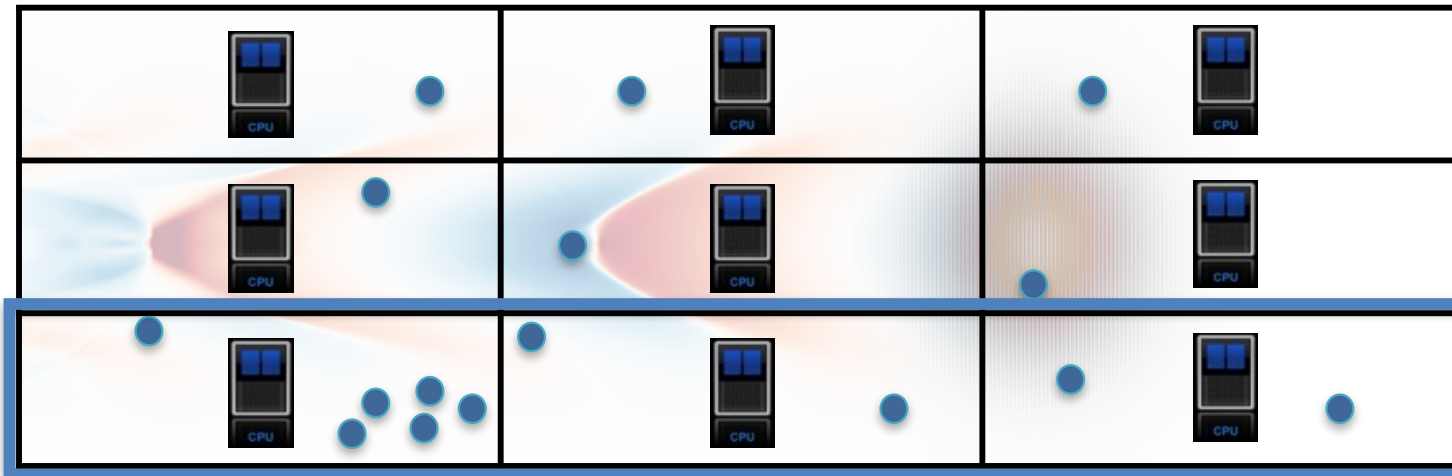
```
from mpi4py.MPI import COMM_WORLD as comm
import numpy as np
```

```
if comm.rank == 0:
    # Create an array
    x = np.ones(10)
    # Sending to rank 1
    comm.Send( x, 1 )
```

```
if comm.rank == 1:
    # Allocate empty array
    x = np.empty(10)
    # Receive the array from 0
    comm.Recv( x, 0 )
    # Print the new x
    print x
```



# Problem: load balancing



Example:  
particle pusher,  
schematic view

Push particle 1	Push particle 1	Push particle 1
Push particle 2	Push particle 2	Push particle 2
Push particle 3	Exchange particles $\longleftrightarrow$	Exchange particles
Push particle 4	Wait	...
Push particle 5	Wait	
Push particle 6	Wait	
Exchange particles $\longleftrightarrow$	Exchange particles	

The simulation will always progress at the pace of the slowest node (the one doing the most work)

=> Problematic when the particle distribution is very non-uniform



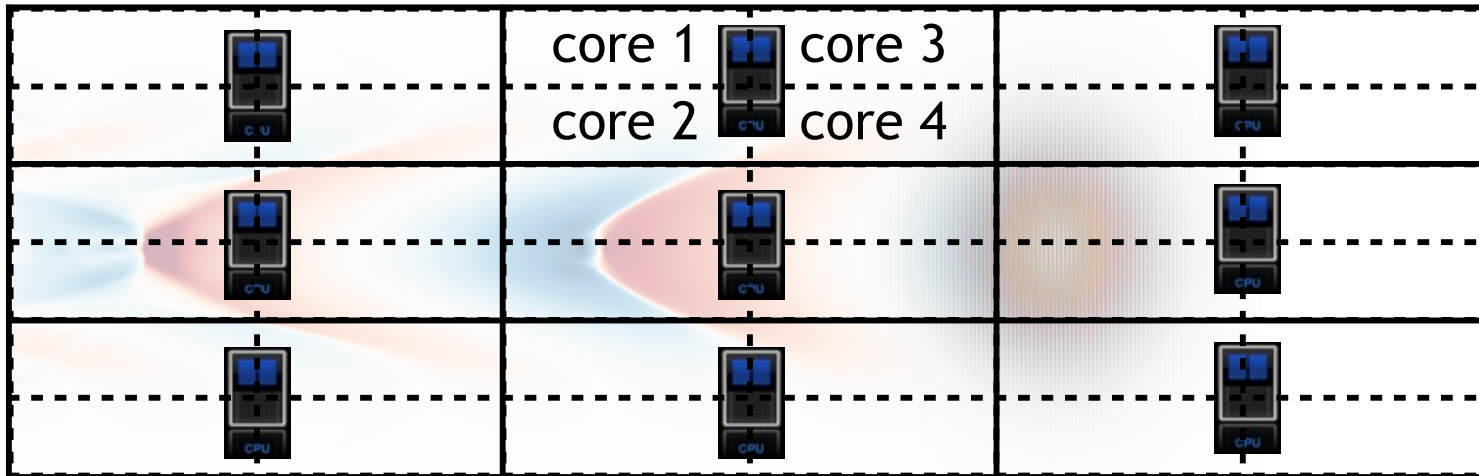
# Outline

- Modern parallel architectures
- Parallelization between nodes: MPI
- Parallelization within one node: OpenMP



# Parallelization within one node: MPI

Parallelization within one node can be done with **MPI**:  
Divide each nodes's sub-domain into smaller sub-domain.  
Each core is tied to one of the smaller sub-domain.



The cores still exchange information via MPI send/receive, within one node.  
(but in this case the information does not go through the network).





# Example: Warp with MPI within one node

Here: one node = one laptop/desktop computer, with several cores

**Compilation (more info on [warp.lbl.gov](http://warp.lbl.gov)):**

Install MPI and mpi4py

Create the files `Makefile.local.pympi` and `setup.parallel.py`

Then type: `make pinstall`

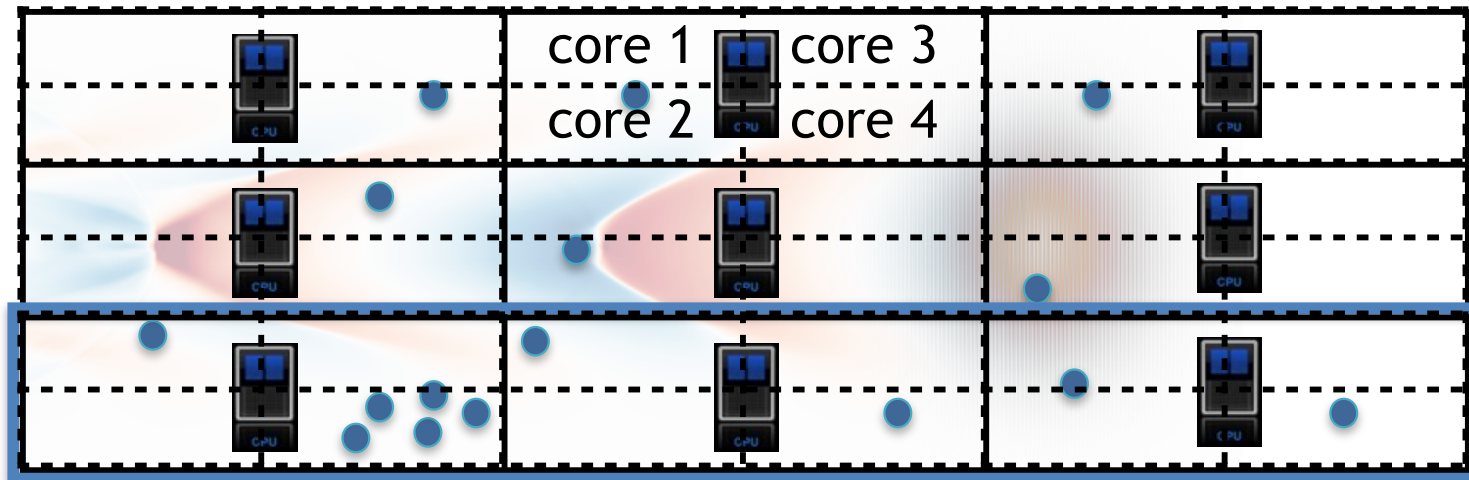
**Running the code:**

```
mpirun -np 4 python warp_script.py
```

(Demonstration)



# MPI within one node: even worse load balancing!

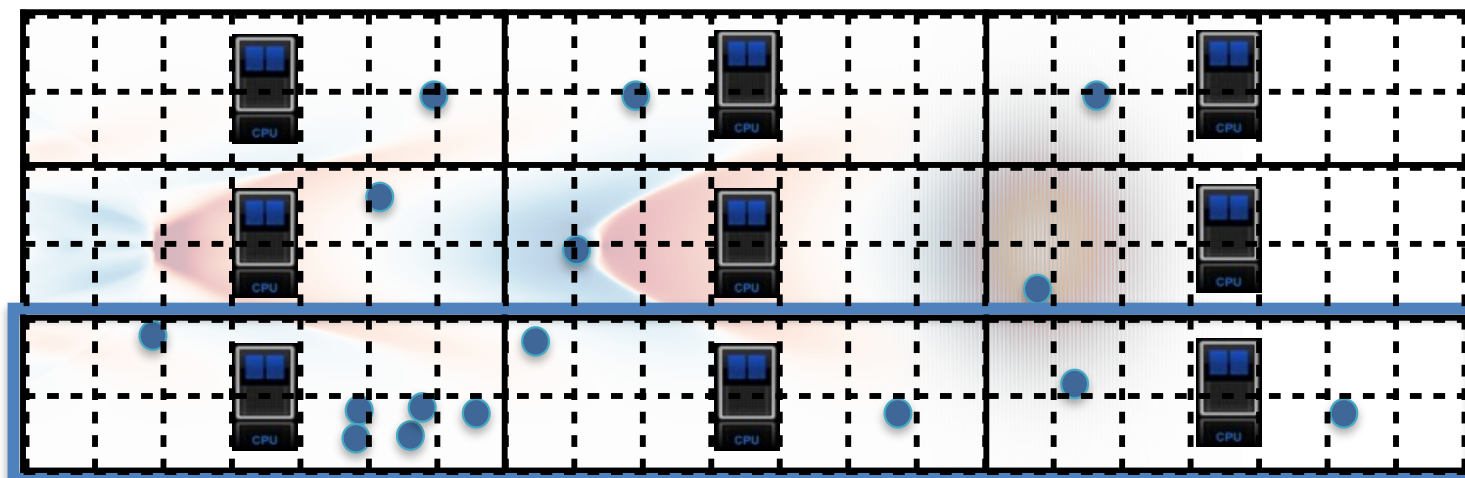


core1	core2	core3	core4	core1	core2	core3	core4	core1	core2	core3	core4
push			push	push			push	push			push
			push	Exchange particles $\longleftrightarrow$ Exchange particles							
			push								
			push								
			push								
Exchange particles $\longleftrightarrow$ Exchange particles											



# OpenMP within one node: load balancing

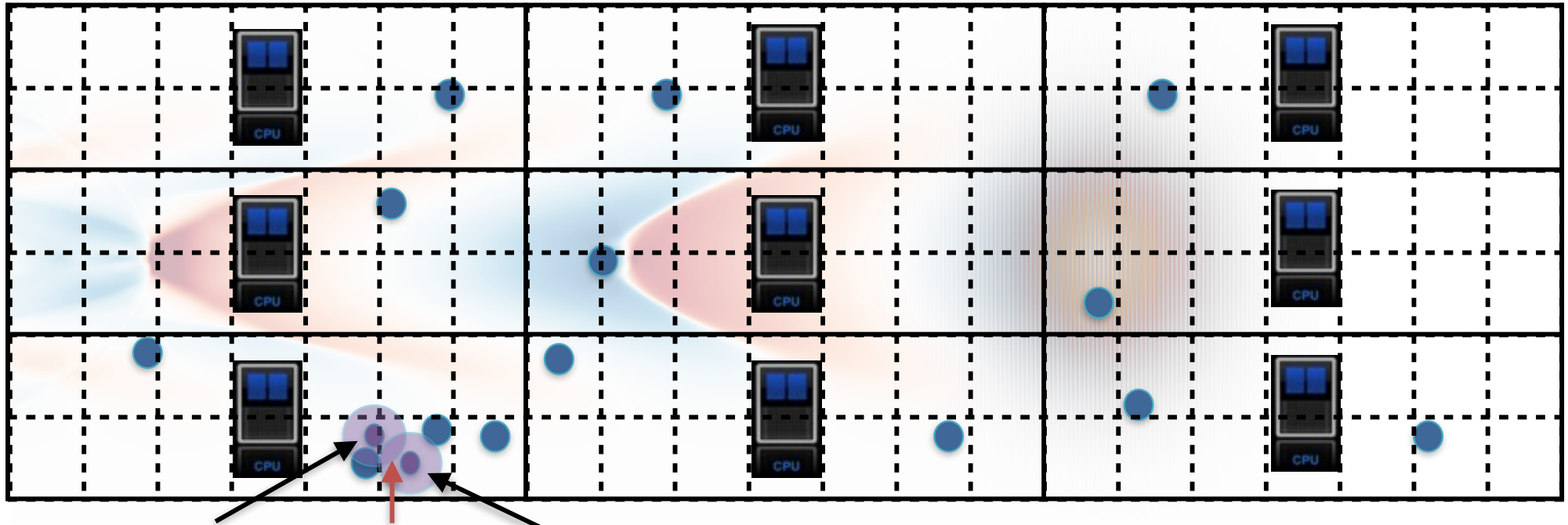
- Create more subdomains than cores  
(here 4 cores per node but 14 subdomains or “tiles”)
- With OpenMP, cores are not tied to one subdomain  
Cores can work one subdomain and then switch to another depending on the work that remains to be done.  
Only possible within one node, because memory is shared



core1	core2	core3	core4	core1	core2	core3	core4	core1	core2	core3	core4
push	push	push	push	push			push	push			push
	push	push		Exchange particles ↔				Exchange particles			
Exchange particles ↔				Exchange particles							



# OpenMP's dangers: race condition



Core 2 performs current deposition **Race condition!** Core 3 performs current deposition

- The cores do not exchange information via MPI send/receive. Instead they **directly** modify the value of the current in shared memory, without notifying the other cores.
- Potentially, two cores could simultaneously try to modify the value of the current in a given cell (leads to inconsistencies). This can be avoided with proper care (e.g. “atomic operations”).



# OpenMP: practical consideration

- **On the developer side:**

Not available in Python, but available in **C** and **Fortran**

Requires to use “**pragmas**” in the code.

Example in Fortran:

```
!!$OMP PARALLEL DO
DO it=1,nt
    ...
    ...
ENDDO
```

(Use OpenMP to do the loop in parallel)  
(Loop over “tiles”)

(Perform work on one “tile”)

- **Warp does not use OpenMP for the moment**

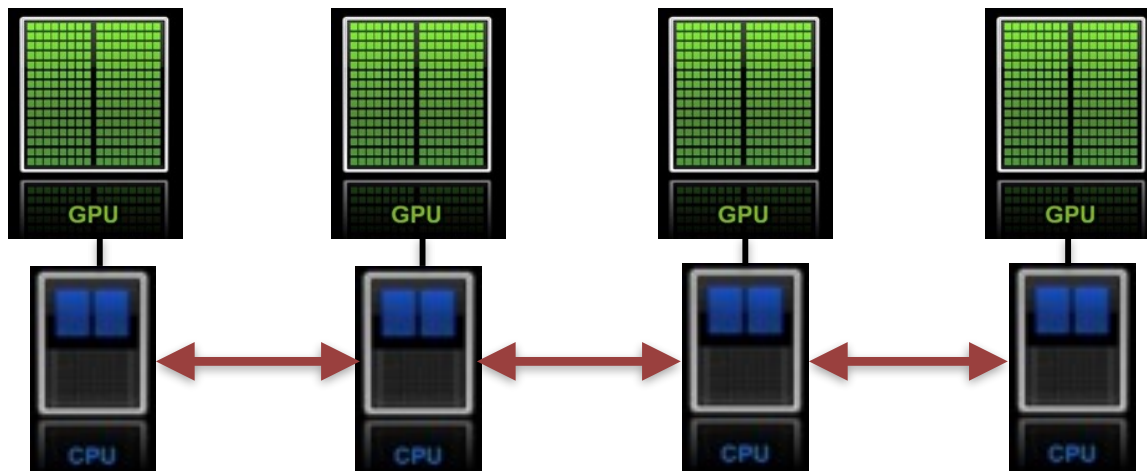
- **But Warp can use PICSAR, which does use OpenMP**

PICSAR = highly-optimized library for elementary operations, such as particle pusher, current deposition, field gathering, etc. PICSAR is soon to be released as open-source.



# GPU programming

- **Conceptual similarities with OpenMP programming:**  
load balancing by tiling, race conditions
- **But also differences:**
  - ~1000s (slow) cores instead of 10-60 cores
  - Only connected to the network through an associated CPU
  - GPU programming uses specific language (CUDA, OpenCL, ...)
- **The trend for the future is to bridge the difference between many-core CPUs and GPU:**  
hardware (more cores on CPU, GPUs to be integrated with CPUs)  
language (OpenMP starts targeting GPUs)



# Summary

- Parallel architectures are organized around (at least) **two levels of parallelization**:
  - Inter-nodes (uses network)
  - Intra-node (uses shared memory)
- The “traditional” paradigm (in the PIC community) is to use MPI at both levels. This is limited, esp. due to load-balancing.
- “Novel” paradigms are becoming more and more common: MPI+OpenMP (with tiles), MPI+GPU, etc.

